Chapter 11: Database Access

Objectives

- Understand the Python DB API architecture
- Connect to a database
- Execute simple and parameterized queries
- Fetch single and multiple row results
- Get metadata about a query
- Execute non-query statements
- Start transactions and commit or rollback as needed

The DB API

- Several ways to access DBMSs from Python
- DB API is most popular
- DB API is sort of an "abstract class"
- Many modules for different DBMSs using DB API
- Hides actual DBMS implementation

To make database programming simpler, Python has the DB API. This is an API to standardize working with databases. When a package is written to access a database, it is written to conform to the API, and thus programmers do not have to learn a new set of methods and functions.

Table 14. Available Interfaces (using Python DB API-2.0)

Database	Python package
Firebird (and Interbase)	KInterbasDB
IBM DB2	PyDB2
Informix	informixdb
Ingres	ingmod
Microsoft SQL Server	pymssql
MySQL	pymysql
ODBC	pyodbc
Oracle	cx_oracle
PostgreSQL	psycopg
SAP DB (also known as "MaxDB")	sapdbapi
SQLite	sqlite3
Sybase	Sybase

There may be other interfaces to some of the listed DBMSs as well.

Connecting to a Server

- Import appropriate library
- Use connect() to get a database object
- Specify host, database, username, password

To connect to a database server, import the package for the specific database. Use the package's **connect()** method to get a database object, specifying the host, initial database, username, and password. If the username and password are not needed, use None.

Some database modules have nonstandard parameters to the connect() method.

When finished with the connection, call the **close()** method on the connection object.

Many database modules support the context manager (**with** statement), and will automatically close the database when the with block is exited. Check the documentation to see how this is implemented for a specific database.

Example

```
import sqlite3

slconn = sqlite3.connect('web_content')

import pymysql

myconn = pymysql.connect (host = "myserver1",
                          user = "adeveloper",
                          passwd = "s3cr3t",
                          db = "web_content")
#  make queries, etc. here ...
myconn.close()
```

Argument names for the connect() method may not be consistent. For instance, pymysql supports the above parameter names, while pymssql does not.

Table 15. connect() examples

Package	Database	Connection
cx_oracle	Oracle	ip = 'localhost' + port = 1521 + SID = 'YOURSIDHERE' + dsn_tns = cx_Oracle.makedsn(ip, port, SID) + + db = cx_Oracle.connect('adeveloper', '$3cr3t', dsn_tns)
psychopg	PostgreSQL	psycopg2.connect ("' + host='localhost' + user='adeveloper' + password='$3cr3t' + dbname='testdb' + "') *note connect() has one parameter: a string, not multiple parameters*
pymssql	MS-SQL	pymssql.connect (+ host="localhost", + user="adeveloper", + passwd="$3cr3t", + db="testdb", +) + pymssql.connect (+ dsn="DSN", +)
pymysql	MySQL	pymysql.connect (+ host="localhost", + user="adeveloper", + passwd="$3cr3t", + db="testdb", +)
pyodbc	Any ODBC-compliant DB	pyodbc.connect("' + DRIVER={SQL Server}; + SERVER=localhost; + DATABASE=testdb; + UID=adeveloper; + PWD=$3cr3t + "') + pyodbc.connect('DSN=testdsn;PWD=$3cr3t')
sqlite3	SqlLite3	sqlite3.connect('testdb') + sqlite3.connect(':memory:')

Creating a Cursor

- Cursor is object that can execute SQL statements
- Several kinds of cursors usually available
- Other cursors leave data on server, etc.

Once you have a database object, you can create one or more cursors. A cursor is an object that can execute SQL code and fetch results.

The default cursor for most packages returns each row as a tuple of values. There are different types of cursors that can return data in different formats, or that control whether data is stored on the client or the server.

Example

```
myconn = pymysqlconnect (host="myserver1",user="adeveloper",
passwd="s3cr3t", db="web_content")
mycursor = myconn.cursor()
```

Executing a Statement

- Executing cursor sends SQL to server
- Data not returned until asked for
- Returns number of lines in result set for queries
- Returns lines affected for other statements

Once you have a cursor, you can use it to perform queries, or to execute arbitrary SQL statements via the execute() method. The first argument to execute() is a string containing the SQL statement to run.

Example

```
cursor.execute("select hostname,ostype,user from hostinfo")
cursor.execute('insert into hostinfo values
("foo",5,"2.6","arch","net",2055,3072,"bob",0)')
```

Fetching Data

- Use one of the fetch methods from the cursor object
- Syntax
  - rec = cursor.fetchone()
  - recs = cursor.fetchall()
  - recs = cursor.fetchmany()

Cursors provide three methods for returning query results.

fetchone() returns the next available row from the query results.

fetchall() returns a tuple of all rows.

fetchmany(n) returns up to n rows. This is useful when the query returns a large number of rows.

Example

```
cursor.execute("select color, quest from knights where name = 'Robin'")
(color,quest) = cursor.fetchone()

cursor.execute("select color, quest from knights")
rows = cursor.fetchall()

cursor.execute("select * from huge_table)
while True:
    rows = cursor.fetchmany(1000)
    if rows == []:
        break
    for row in rows:
        # process row
```

Example

db_sqlite_basics.py

```
#!/usr/bin/env python

import sqlite3

with sqlite3.connect("../DATA/PRESIDENTS") as s3conn:

    s3cursor = s3conn.cursor()

    # select first name, last name from all presidents
    s3cursor.execute('''
        select lname, fname
        from presidents
    ''')

    print("Sqlite3 does not provide a row count\n")

    for row in s3cursor.fetchall():
        print(' '.join(row))
    # note -- Sqlite3 does not return the number of rows
    print()
```

db_sqlite_basics.py

```
Nixon Richard Milhous
Ford Gerald Rudolph
Carter James Earl 'Jimmy'
Reagan Ronald Wilson
Bush George Herbert Walker
Clinton William Jefferson 'Bill'
Bush George Walker
Obama Barack Hussein
Trump Donald J
```

See db_mysql_basics.py and db_postgres_basics.py for examples using those modules.

Parameterized Statements

- More efficient updates
- Use cursor.execute() or cursor.executemany()
- Use placeholders in query
- Pass iterable of parameters

For efficiency, you can iterate over of sequence of datasets when performing a non-query SQL statement. The execute() method takes a query, plus an iterable of values to fill in the placeholders. The database manager will only parse the query once, then reuse it for subsequent calls to execute().

Different database modules use different placeholders. To see what kind of placeholder a module uses, check MODULE.paramstyle. Types include 'format', meaning '%s', and 'qmark', meaning '?'.

The executemany() method takes a query, and an iterable of iterables. It will call execute() once for each nested iterable.

Example

```
singledata = ("Smith","John","green"),

multidata = [
    ("Smith","John","green"),
    ("Douglas","Sam","pink"),
    ("Robinson","Alberta","blue"),
]

query = "insert into people (lname,fname,color) values (%s,%s,%s)"

rows_added = cursor.execute(query, singledata)
rows_added = cursor.executemany(query, multidata)
```

Table 16. Placeholders for SQL Parameters

Python package	Placeholder for parameters
pymysql	%s
cx_oracle	:param_name
pyodbc	?
pgdb	%s
pymssql	%d int %s str etc.
Psychopg	%s or %(param_name)s
sqlite3	?

TIP with the exception of **pymssql** the same placeholder is used for all column types.

Example

db_sqlite_parameterized.py

```
#!/usr/bin/env python

import sqlite3

with sqlite3.connect("../DATA/PRESIDENTS") as s3conn:

    s3cursor = s3conn.cursor()

    party_query = '''
    select lname, fname
    from presidents
        where party = ?
    '''

    for party in 'Federalist', 'Whig':
        print(party)
        s3cursor.execute(party_query, (party,))
        print(s3cursor.fetchall())
        print()
```

db_sqlite_parameterized.py

```
Federalist
[('Adams', 'John')]

Whig
[('Harrison', 'William Henry'), ('Tyler', 'John'), ('Taylor', 'Zachary'),
('Fillmore', 'Millard')]
```

Dictionary Cursors

- Indexed by column name
- Not standardized in the DB API

The standard cursor provided by the DB API returns a tuple for each row. Most DB packages provide other kinds of cursors, including user-defined versions.

A very common cursor is a dictionary cursor, which returns a dictionary for each row, where the keys are the column names. Each package that provides a dictionary cursor has its own way of providing the dictionary cursor, although they all work the same way.

For the packages that don't have a dictionary cursor, you can make a generator function that will emulate one.

Table 17. Dictionary Cursors

Python package	**How to get a dictionary cursor**
pymysql	import pymysql.cursors conn = pymysql.connect(..., cursorclass = pymysql.cursors.DictCursor) dcur = conn.cursor() all cursors will be dict cursors or dcur = conn.cursor(pymysql.cursors.DictCursor) only this cursor will be a dict cursor
cx_oracle	Not available directly, but can be emulated using columns from CUR.description
pyodbc	Not available directly, but can be emulated using columns from CUR.description
pgdb	Not available directly, but can be emulated using columns from CUR.description
pymssql	conn = pymssql.connect (..., as_dict=True) dcur = conn.cursor()
psychopg	import psycopg2. extras dcur = conn.cursor(cursor_factory=psycopg.extras.DictCursor)
sqlite3	conn = sqlite3.connect (..., row_factory=sqlite3.Row) dcur = conn.cursor() or conn.row_factory = sqlite3.Row dcur = conn.cursor()

Example

db_sqlite_extras.py

```
#!/usr/bin/env python

import sqlite3

s3conn = sqlite3.connect("../DATA/PRESIDENTS")
# uncomment to make _all_ cursors dictionary cursors
# conn.row_factory = sqlite3.Row

NAME_QUERY = '''
    select lname, fname
    from presidents
    where num < 5
'''

cur = s3conn.cursor()

# select first name, last name from all presidents
cur.execute(NAME_QUERY)

for row in cur.fetchall():
    print(row)
print('-' * 50)

dcur = s3conn.cursor()

# make _this_ cursor a dictionary cursor
dcur.row_factory = sqlite3.Row

# select first name, last name from all presidents
dcur.execute(NAME_QUERY)

for row in dcur.fetchall():
    print(row['fname'], row['lname'])

print('-' * 50)
```

db_sqlite_extras.py

```
('Washington', 'George')
('Adams', 'John')
('Jefferson', 'Thomas')
('Madison', 'James')
--------------------------------------------------
George Washington
John Adams
Thomas Jefferson
James Madison
--------------------------------------------------
```

Metadata

- cursor.description returns tuple of tuples
- Fields
  - name
  - type_code
  - display_size
  - internal_size
  - precision
  - scale
  - null_ok

Once a query has been executed, the cursor's description() method returns information about the query as a tuple of tuples.

There is one tuple for each column in the query; each tuple contains a tuple of 7 values describing the column.

For instance, to get the names of the columns, you could say:

```
names = [ d[0] for d in cursor.description ]
```

For non-query statements, cursor.description returns None.

Tip: The names are based on the query, and not necessarily on the 'official' names in the database itself.

Example

db_sqlite_emulate_dict_cursor.py

Note: Most database modules, including pymysql, have a dictionary cursor built in — this is just for an example you could use with any DB API module that does not have this capability. This uses the metadata from the cursor to get the column names, and forms a dictionary by zipping the column names with the column values. Another approach would be to use a named tuple.

```
#!/usr/bin/env python

import sqlite3

s3conn = sqlite3.connect("../DATA/PRESIDENTS")

c = s3conn.cursor()

def row_as_dict(cursor):
    '''Generate rows as dictionaries'''
    column_names = [desc[0] for desc in cursor.description]
    for row in cursor.fetchall():
        row_dict = dict(zip(column_names, row))
        yield row_dict


# select first name, last name from all presidents
num_recs = c.execute('''
    select lname, fname
    from presidents
''')

for row in row_as_dict(c):
    print(row['fname'], row['lname'])
```

db_sqlite_emulate_dict_cursor.py

```
Lyndon Baines Johnson
Richard Milhous Nixon
Gerald Rudolph Ford
James Earl 'Jimmy' Carter
Ronald Wilson Reagan
George Herbert Walker Bush
William Jefferson 'Bill' Clinton
George Walker Bush
Barack Hussein Obama
Donald J Trump
```

See db_sqlite_named_tuple_cursor.py for a similar example that creates named tuples rather than dictionaries for each row.

Transactions

- Transactions allow safer control of updates
- commit() to save transactions
- rollback() to discard
- Default is autocommit off
- autocommit\=True to turn on

To be certain of data integrity, you may want to make batch changes to your database and commit the changes only when successful. By default, you must explicitly save your data using connection.commit().

To save all transactions for a database since last commit or rollback, use connection.commit(). To discard transactions, use connection.rollback().

You can also turn on autocommit, which calls commit() after every statement that modifies the database.

Example

```
try:
    for info in list_of_tuples:
         cursor.execute(query,info)
except SQLError:
    dbconn.rollback()
else:
    dbconn.commit()
```

pymysql only supports transaction processing for tables using the InnoDB engine

Object-relational Mappers

- No SQL required
- Maps a class to a table
- All DB work is done by manipulating objects
- Most popular Python ORMs
  - SQLAlchemy
  - Django (which is a complete web framework)

An Object-relational mapper is a module or framework that creates a level of abstraction above the actual database tables and SQL queries. As the name implies, a Python class (object) is mapped to the actual table.

The two most popular Python ORMs are SQLAlchemy which is a standalone ORM, and Django, which is a very large and complete Web development framework, which provides an ORM as a subpackage. SQLAlchemy is the most fully developed package, and is the ORM used by Flask and other Web development frameworks.

Instead of querying the database, you call a search method on an object representing a table. To add a row to the table, you create a new instance of the table class, populate it, and call a method like save(). You can create a large, complex database system, complete with foreign keys, composite indices, and all the other attributes near and dear to a DBA, without writing the first line of SQL.

You can use Python ORMs in two ways.

One way is to design the database with the ORM. To do this, you create a class for each table in the database, specifying the columns with predefined classes from the ORM. Then you run an ORM command which executes the queries needed to build the database. If you need to make changes, you update the class definitions, and run an ORM command to synchronize the actual DBMS to your classes.

The second way is to map tables to an existing database. You create the classes to match the schemas that have already been defined in the database. Both SQLAlchemy and the Django ORM have tools to automate this process.

Chapter 11 Exercises

Exercise 11-1 (presidentmysql.py, presidentsqlite.py)

For this exercise, you can use the SQLite3 database provided, or use your own DBMS. The mkpres.sql script is generic and should work with any DBMS to create and populate the presidents table. The SQLite3 database is named PRESIDENTS and is created in the DATA folder of the student files.

The data has the following layout

Table 18. Layout of President Table

Field Name	Data Type	Null	Default
num	int(11)	YES	NULL
lname	varchar(32)	YES	NULL
fname	varchar(64)	YES	NULL
dstart	date	YES	NULL
dend	date	YES	NULL
birthplace	varchar(128)	YES	NULL
dbirth	date	YES	NULL
ddeath	date	YES	NULL
party	varchar(32)	YES	NULL

Refactor your **president.py** module to get its data from this table, rather than from a file. Re-run your previous scripts that used president.py; now they should get their data from the database, rather than from the flat file.

Exercise 11-2 (add_pres_mysql.py, add_pres_sqlite.py)

Assuming your favorite candidate wins (or has won) the next election, add them to the table.

HINT: If you don't have access to the Internet, make up their information!

SQL syntax for adding a record is

```
INSERT INTO table ("COL1-NAME",..) VALUES ("VALUE1",...)
```

To do a parameterized insert (the right way!):

```
INSERT INTO table ("COL1-NAME",..) VALUES (%s,%s,...)  # MySQL
INSERT INTO table ("COL1-NAME",..) VALUES (?,?,...)    # SQLite
```

or whatever your database uses as placeholders