# Python Database Access

John Strickler

Version 1.0, March 2021

# Table of Contents

# Chapter 1: 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 1. 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 | psycopg2 |
| SAP DB (also known as "MaxDB") | sapdbapi |
| SQLite | sqlite3 |
| Sybase | Sybase |

**NOTE** 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()
```

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

*Table 2. connect() examples*

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

# Creating a Cursor

- Cursor can execute SQL statements
- Multiple cursors available
  - Standard cursor
    - Returns tuples
  - Other cursors
    - Returns dictionaries
    - Leaves data on server

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 not rows:
        break
    for row in rows:
        # process row
```

## Example

**db_sqlite_basics.py**

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

import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn:  ①

    s3cursor = s3conn.cursor()  ②

    # select first name, last name from all presidents
    s3cursor.execute('''
        select firstname, lastname
        from presidents
    ''')  ③

    print("Sqlite3 does not provide a row count\n")  ④

    for row in s3cursor.fetchall():  ⑤
        print(' '.join(row))  ⑥
```

① connect to the database

② get a cursor object

③ execute a SQL statement

④ (included for consistency with other DBMS modules)

⑤ fetchall() returns all rows

⑥ each row is a tuple

*db_sqlite_basics.py*

```
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
Joseph Robinette Biden
```

**NOTE**

See db_mysql_basics.py and db_postgres_basics.py for examples using those modules. In general, all the sqlite3 examples are also implemented for MySQL and Postgres, plus a few extras.

# SQL Injection

- "Hijacks" SQL code
- Result of string formatting
- Always use parameterized statements

One kind of vulnerability in SQL code is called SQL injection. This occurs when an attacker embeds SQL commands in input data. This can happen when naively using string formatting to build SQL statements:

## Example

**db_sql_injection.py**

```
#!/usr/bin/env python
#
good_input = 'Google'
malicious_input = "'; drop table customers; -- "  ①

naive_format = "select * from customers where company_name = '{}' and company_id != 0"

good_query = naive_format.format(good_input)  ②
malicious_query = naive_format.format(malicious_input)  ②

print("Good query:")
print(good_query)  ③
print()

print("Bad query:")
print(malicious_query)  ④
```

① input would come from a web form, for instance

② string formatting naively adds the user input to a field, expecting only a customer name

③ non-malicious input works fine

④ query now drops a table ('--' is SQL comment)

*db_sql_injection.py*

```
Good query:
select * from customers where company_name = 'Google' and company_id != 0

Bad query:
select * from customers where company_name = ''; drop table customers; -- ' and
company_id != 0
```

# Parameterized Statements

- More efficient updates
- Use placeholders in query
  - Placeholders vary by DB
- Pass iterable of parameters
- Prevent SQL injection
- Use cursor.execute() or cursor.executemany()

For efficiency, you can iterate over of sequence of input 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().

Parameterized queries also protect against SQL injection attacks.

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

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

## Example

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

multi_rows= [
    ("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, single_row)
rows_added = cursor.executemany(query, multi_rows)
```

*Table 3. Placeholders for SQL Parameters*

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

**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.db") as s3conn:
    s3cursor = s3conn.cursor()

    party_query = '''
    select firstname, lastname
    from presidents
        where party = ?
    '''  ①

    for party in 'Federalist', 'Whig':
        print(party)
        s3cursor.execute(party_query, (party,))  ②
        print(s3cursor.fetchall())
        print()
```

① ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use different placeholders

② second argument to execute() is iterable of values to fill in placeholders from left to right

*db_sqlite_parameterized.py*

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

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

## Example

**db_sqlite_bulk_insert.py**

```
#!/usr/bin/env python
import os
import sqlite3
import random

FRUITS = ["pomegranate", "cherry", "apricot", "date", "apple",
          "lemon", "kiwi", "orange", "lime", "watermelon", "guava",
          "papaya", "fig", "pear", "banana", "tamarind", "persimmon",
          "elderberry", "peach", "blueberry", "lychee", "grape"]

DB_NAME = 'fruitprices.db'  ①

CREATE_TABLE = """
create table fruit (
    name varchar(30),
    price decimal
)
"""  ②

INSERT = '''
insert into fruit (name, price) values (?, ?)
'''  ③


def main():
    """
    Program entry point.

    :return: None
    """
    conn = get_connection()
    create_database(conn)
    populate_database(conn)

    read_database()


def get_connection():
    """
    Get a connection to the PRODUCE database

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
```

```
        os.remove(DB_NAME)  ④

    s3conn = sqlite3.connect(DB_NAME)  ⑤
    return s3conn


def create_database(conn):
    """
    Create the fruit table

    :param conn: The database connection
    :return: None
    """
    conn.execute(CREATE_TABLE)  ⑥


def populate_database(conn):
    """
    Add rows to the fruit table

    :param conn: The database connection
    :return: None
    """

    fruit_data = get_fruit_data()  # [('apple', .49), ('kiwi', .38)]

    try:
        conn.executemany(INSERT, fruit_data)  ⑦
    except sqlite3.DatabaseError as err:
        print(err)
        conn.rollback()
    else:
        conn.commit()  ⑧


def get_fruit_data():
    """
    Create iterable of fruit records.

    :return: Generator of name/price tuples.
    """
    return ((f, round(random.random() * 10 + 5, 2)) for f in FRUITS)  ⑨


def read_database():
    conn = sqlite3.connect(DB_NAME)
    for name, price in conn.execute('select name, price from fruit'):
        print('{:12s} {:6.2f}'.format(name, price))
```

```
if __name__ == '__main__':
    main()
```

① set name of database

② SQL statement to create table

③ parameterized SQL statement to insert one record

④ remove existing database if it exists

⑤ connect to (new) database

⑥ run SQL to create table

⑦ iterate over list of pairs and add each pair to the database

⑧ commit the inserts; without this, no data would be saved

⑨ build list of tuples containing fruit, price pairs

*db_sqlite_bulk_insert.py*

```
pomegranate   12.85
cherry         8.66
apricot       10.49
date           8.19
apple          5.44
lemon          5.53
kiwi           9.58
orange         8.50
lime          10.01
watermelon    10.45
guava         12.68
papaya        12.92
fig           12.37
pear          11.82
banana        13.42
tamarind       8.01
persimmon      9.53
elderberry     5.72
peach         13.19
blueberry     10.10
lychee        12.75
grape          8.62
```

# 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 4. Dictionary Cursors*

| Python package | How to get a dictionary cursor |
|---|---|
| pymysql | `import pymysql.cursors`<br>`conn = pymysql.connect(…,`<br>`cursorclass = pymysql.cursors.DictCursor`<br>`)`<br>`dcur = conn.cursor()`<br>*all cursors will be dict cursors*<br><br>`dcur = conn.cursor( pymysql.cursors.DictCursor)`<br>*only this cursor will be a dict cursor* |
| cx_oracle | *Not available* |
| pyodbc | *Not available* |
| pgdb | *Not available* |
| pymssql | `conn = pymssql.connect (…, as_dict=True)`<br>`dcur = conn.cursor()` |
| psychopg | `import psycopg2.extras`<br>`dcur =`<br>`conn.cursor(cursor_factory=psycopg.extras.DictCursor)` |
| sqlite3 | `conn = sqlite3.connect (…,`<br>`row_factory=sqlite3.Row)`<br>`dcur = conn.cursor()`<br>`conn.row_factory = sqlite3.Row`<br>`dcur = conn.cursor()` |

## Example

**db_sqlite_extras.py**

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

import sqlite3

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

NAME_QUERY = '''
    select firstname, lastname
    from presidents
    where termnum < 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()  ①

# make _this_ cursor a dictionary cursor
dcur.row_factory = sqlite3.Row  ②

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

for row in dcur.fetchall():
    print(row['firstname'], row['lastname'])  ③

print('-' * 50)
```

① default cursor returns tuple for each row

② Row object is tuple/dict hybrid; can be indexed by position OR column name

③ selecting by column name

*db_sqlite_extras.py*

```
('George', 'Washington')
('John', 'Adams')
('Thomas', 'Jefferson')
('James', 'Madison')
------------------------------------------------
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 attribute is a tuple with metadata about the columns in the query. It contains one tuple for each column in the query, containing 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.

The names are based on the query (with possible aliases), and not necessarily on the names in the table.

## Example

| | |
|---|---|
| **NOTE** | Many 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. The example 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. __ |

**db_sqlite_emulate_dict_cursor.py**

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

import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")

c = s3conn.cursor()


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


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

for row in row_as_dict(c):
    print(row['firstname'], row['lastname'])
```

*db_sqlite_emulate_dict_cursor.py*

```
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
Joseph Robinette Biden
```

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

Sometimes a database task involves more than one change to your database (i.e., more than one SQL statement). You don't want the first SQL statement to succeed and the second to fail; this would leave your database in a corrupt state.

To be certain of data integrity, use **transactions**. This lets you make multiple changes to your database and only commit the changes if all the SQL statements were successful. For all packages using the Python DB API, a transaction is started when you connect. At any point, you can call *CONNECTION*.commit() to save the changes, or *CONNECTION*.rollback() to discard the changes. For most packages, if you don't call commit() after modify a table, the data will not be saved.

**NOTE** You can also turn on autocommit, which calls commit() after every statement.

## Example

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

**NOTE** **pymysql** only supports transaction processing when 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 ORM. Django is a comprehensive 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 some 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.

# NoSQL

- Non-relational database
- Document-oriented
- Can be hierarchical (nested)
- Examples
  - MongoDB
  - Cassandra
  - Redis

A current trend in data storage are called "NoSQL" or non-relational databases. These databases consist of *documents*, which are indexed, and may contain nested data.

NoSQL databases don’t contain tables, and do not have relations.

While relational databases are great for tabular data, they are not as good a fit for nested data. Geo-spatial, engineering diagrams, and molecular modeling can have very complex structures. It is possible to shoehorn such data into a relational database, but a NoSQL database might work much better. Another advantage of NoSQL is that it can adapt to changing data structures, without having to rebuild tables if columns are added, deleted, or modified.

Some of the most common NoSQL database systems are MongoDB, Cassandra and Redis.

## Example

**mongodb_example.py**

```
#!/usr/bin/env python
import re
from pymongo import MongoClient, errors

FIELD_NAMES = (
    'termnumber lastname firstname '
    'birthdate '
    'deathdate birthplace birthstate '
    'termstartdate '
    'termenddate '
    'party'
).split()  ①

mc = MongoClient()  ②

try:
    mc.drop_database("presidents")  ③
except errors.PyMongoError as err:
    print(err)

db = mc["presidents"]  ④

coll = db.presidents  ⑤

with open('../DATA/presidents.txt') as presidents_in:  ⑥
    for line in presidents_in:
        flds = line[:-1].split(':')
        kvpairs = zip(FIELD_NAMES, flds)
        record_dict = dict(kvpairs)
        coll.insert_one(record_dict)  ⑦

print(db.list_collection_names())  ⑧
print()

abe = coll.find_one({'termnumber': '16'})  ⑨
print(abe, '\n')

for field in FIELD_NAMES:
    print("{0:15s} {1}".format(field.upper(), abe[field]))  ⑩

print('-' * 50)

for president in coll.find():  ⑪
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
```

```
print('-' * 50)

rx_lastname = re.compile('^roo', re.IGNORECASE)
for president in coll.find({'lastname': rx_lastname}):  ⑫
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

for president in coll.find({"birthstate": 'Virginia'}):  ⑬
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))

print('-' * 50)
print("removing Millard Fillmore")
result = coll.delete_one({'lastname': 'Fillmore'})  ⑭
print(result)
result = coll.delete_one({'lastname': 'Roosevelt'})  ⑭
print(result)
print('-' * 50)


result = coll.delete_one({'lastname': 'Bush'})
print(dir(result))
print()

result = coll.count_documents({})  ⑮
print(result)


for president in coll.find():  ⑪
    print("{0[firstname]:25s} {0[lastname]:30s}".format(president))
print('-' * 50)

animals = db.animals

print(animals, '\n')

animals.insert_one({'name': 'wombat', 'country': 'Australia'})
animals.insert_one({'name': 'ocelot', 'country': 'Mexico'})
animals.insert_one({'name': 'honey badger', 'country': 'Iran'})

for doc in animals.find():
    print(doc['name'])
```

① define some field name

② get a Mongo client

③ delete 'presidents' database if it exists

④ create a new database named 'presidents'

⑤ get the collection from presidents db

⑥ open a data file

⑦ insert a record into collection

⑧ get list of collections

⑨ search collection for doc where termnumber == 16

⑩ print all fields for one record

⑪ loop through all records in collection

⑫ find record using regular expression

⑬ find record searching multiple fields

⑭ delete record

⑮ get count of records

*mongodb_example.py*

```
William Howard            Taft
Woodrow                   Wilson
Warren Gamaliel           Harding
Calvin                    Coolidge
Herbert Clark             Hoover
Franklin Delano           Roosevelt
Harry S.                  Truman
Dwight David              Eisenhower
John Fitzgerald           Kennedy
Lyndon Baines             Johnson
Richard Milhous           Nixon
Gerald Rudolph            Ford
James Earl 'Jimmy'        Carter
Ronald Wilson             Reagan
William Jefferson 'Bill'  Clinton
George Walker             Bush
Barack Hussein            Obama
Donald John               Trump
Joseph Robinette          Biden
--------------------------------------------------
Collection(Database(MongoClient(host=['localhost:27017'], document_class=dict,
tz_aware=False, connect=True), 'presidents'), 'animals')

wombat
ocelot
honey badger
```

# Chapter 1 Exercises

## Exercise 1-1 (president_sqlite.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.db** and is located in the DATA folder of the student files.

The data has the following layout

*Table 5. Layout of President Table*

| Field Name | Data Type | Null | Default |
|---|---|---|---|
| termnum | int(11) | YES | NULL |
| lastname | varchar(32) | YES | NULL |
| firstname | varchar(64) | YES | NULL |
| termstart | date | YES | NULL |
| termend | date | YES | NULL |
| birthplace | varchar(128) | YES | NULL |
| birthstate | varchar(32) | YES | NULL |
| birthdate | date | YES | NULL |
| deathdate | date | YES | NULL |
| party | varchar(32) | YES | NULL |

Refactor the **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.

| NOTE | If you created a president.py module as part of an earlier lab, use that. Otherwise, use the supplied president.py module in the top folder of the student files. |
|---|---|

## Exercise 1-2 (add_pres_sqlite.py)

Add the next president to the presidents database. Just make up the data — let's keep this non-political. Don't use any real-life people.

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*

> **NOTE** There are also MySQL versions of the answers.

# Index

**T**



 