DB Programming

John Strickler

Version 1.0, November 2024

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
- Execute non-query statements
- Get metadata about a query
- Start transactions and commit or rollback as needed

The DB API

- Most popular Python DB interface
- Specification, not abstract class
- Many modules for different DB implementations
- Hides actual DBMS implementation

To make database programming simpler and more consistent, Python provides 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 for each different database architecture.

DB API objects and methods

```
conn = package.connect(connection-arguments)
cursor = conn.cursor()
num_lines = cursor.execute(query)
num_lines = cursor.execute(query-with-placeholders, iterable)
num_lines = cursor.executemany(query-with-placeholders, iterable)
all_rows = cursor.fetchall()
some_rows = cursor.fetchmany(n)
one_row = cursor.fetchone()
conn.commit()
conn.rollback()
```


Table 1. Available Interfaces (using Python DB API-2.0)

Database	Python package
Firebird (and Interbase)	KInterbasDB
IBM DB2	ibm-db
Informix	informixdb
Ingres	ingmod
Microsoft SQL Server	pymssql
MySQL	pymysql
ODBC	pyodbc
Oracle	cx_oracle
PostgreSQL	psycopg (*previously* psycopg2)
SAP DB (also known as "MaxDB")	sapdbapi
SQLite	sqlite3
Sybase	Sybase

This list is not comprehensive, and there are additional interfaces available for some of the listed DBMSs.

Connecting to a Server

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

To connect to a database server, import the package for the specific database. Use the package's `connect()` method to get a connection object. The `connect()` function requires the information needed to access the database, which may include the host, initial database, username, or password.

Argument names for the `connect()` method are not consistent across packages. Most `connect()` methods use individual arguments, such as ***host***, ***database***, etc., but some use a single string argument.

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 particular database.

Example

```
import pymysql

conn = pymysql.connect (host = "dbserver",
                        user = "adeveloper",
                        passwd = "s3cr3t",
                        db = "samples")
# Interact with database here ...
conn.close()
```

```
import sqlite3

with sqlite3.connect('sample.db') as conn:
    # Interact with database here ...
```


Table 2. connect() examples

<table>
<tr><th>Database</th><th>Python package</th><th>Connection</th></tr>
<tr><td>IBM DB2</td><td>ibm-db</td><td><pre>import ibm_db_dbi as db2
conn = db2.connect(
    "DATABASE=testdb;HOSTNAME=localhost;PORT=50000;PROTOCOL=TCPIP;UID=db2inst1;PWD=scripts;",
    "",
    ""
)</pre></td></tr>
<tr><td>Oracle</td><td>cx_oracle</td><td><pre>ip = 'localhost'
port = 1521
SID = 'YOURSIDHERE'
dsn_tns = cx_Oracle.makedsn(ip, port, SID)
db = cx_Oracle.connect('adeveloper', '$3cr3t', dsn_tns)</pre></td></tr>
<tr><td>PostgreSQL</td><td>psychopg</td><td><pre>psycopg.connect ('''
    host='localhost'
    user='adeveloper'
    password='$3cr3t'
    dbname='testdb'
''')</pre>
<code>connect()</code> has one <code>str</code> parameter, not multiple parameters</td></tr>
<tr><td>MS-SQL</td><td>pymssql</td><td><pre>pymssql.connect (
    host="localhost",
    user="adeveloper",
    passwd="$3cr3t",
    db="testdb",
)
pymssql.connect (
    dsn="DSN",
)</pre></td></tr>
<tr><td>MySQL</td><td>pymysql</td><td><pre>pymysql.connect (
    host="localhost",
    user="adeveloper",
    passwd="$3cr3t",
    db="testdb",
)</pre></td></tr>
</table>

<table>
<tr><th>Database</th><th>Python package</th><th>Connection</th></tr>
<tr><td>ODBC-compliant DB</td><td>pyodbc</td><td>

```
pyodbc.connect('''
   DRIVER={SQL Server};
   SERVER=localhost;
   DATABASE=testdb;
   UID=adeveloper;
   PWD=$3cr3t
''')

pyodbc.connect('DSN=testdsn;PWD=$3cr3t')
```

`connect()` has one (string) parameter, not multiple parameters
</td></tr>
<tr><td>SqlLite3</td><td>sqlite3</td><td>

```
sqlite3.connect('testdb')   # on-disk database(single file)
sqlite3.connect(':memory:')  # in-memory database
```

</td></tr>
</table>

Creating a Cursor

- Cursor can execute SQL statements
- Create with `cursor()` method
- Multiple cursors available
  - Standard cursor
    - Returns rows as tuples
  - Other cursors
    - Return dictionary
    - Return hybrid dictionary/list
    - Leave data on server

Once you have a connection object, call `cursor()` to create a cursor object. A cursor is an object that can execute SQL code and fetch results. Each connection may have one or more active cursors.

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

> The examples in this chapter are implemented with SQLite, since the `sqlite3` module is part of the standard library. Most of the examples are also implemented for PostgreSQL and MySQL. See `db_mysql_*.py` and `db_postgres_*.py` in EXAMPLES.

Example

```
import sqlite3
conn = sqlite3.connect("sample.db")
cursor = conn.cursor()
```

Querying data

- ***cursor***.execute(***query***)
  - Gets all data from query
  - Returns # rows in result set
- Use **fetch…** methods
  - `.fetchall()`
  - `.fetchone()`
  - `.fetchmany()`
- Return rows as tuples of values

Once you have a cursor, you can use it to execute queries via the `execute()` method. The first argument to `execute()` is a string containing one SQL statement.

For queries, `execute()` returns the number of rows in the result set. For non-query statements, `execute()` returns the number of rows affected by the operation.

For `sqlite3`, `execute()` returns the cursor object, so you can say `execute(QUERY-STATEMENT).fetchall()`.

Fetch methods

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.

For all three methods, each row is returned as a tuple of values.

For standard cursors, all data is transferred from the database server to your program's memory when `execute()` is called.

Example

db_sqlite_basics.py

```python
import sqlite3

# conn = sqlite3.Connection(...)
with sqlite3.connect("../DATA/presidents.db") as conn:  # connect to the database

    s3_cursor = conn.cursor()  # get a cursor object

    # select specified columns from all presidents
    s3_cursor.execute('''
        select termnum, firstname, lastname, party
        from presidents
    ''')  # execute a SQL statement

    for term, firstname, lastname, party in s3_cursor.fetchall():
        print(f"{term:2d} {firstname:25} {lastname:20} {party}")
    print()
```

db_sqlite_basics.py

```
 1 George                     Washington          no party
 2 John                       Adams               Federalist
 3 Thomas                     Jefferson           Democratic - Republican
 4 James                      Madison             Democratic - Republican
 5 James                      Monroe              Democratic - Republican
 6 John Quincy                Adams               Democratic - Republican
 7 Andrew                     Jackson             Democratic
 8 Martin                     Van Buren           Democratic
 9 William Henry              Harrison            Whig
10 John                       Tyler               Whig
11 James Knox                 Polk                Democratic
12 Zachary                    Taylor              Whig
```

…

```
36 Lyndon Baines              Johnson             Democratic
37 Richard Milhous            Nixon               Republican
38 Gerald Rudolph             Ford                Republican
39 James Earl 'Jimmy'         Carter              Democratic
40 Ronald Wilson              Reagan              Republican
41 George Herbert Walker      Bush                Republican
42 William Jefferson 'Bill'   Clinton             Democratic
43 George Walker              Bush                Republican
44 Barack Hussein             Obama               Democratic
45 Donald J                   Trump               Republican
46 Joseph Robinette           Biden               Democratic
```


Non-query statements

- Update database
- Returns count of rows affected
- Changes must be committed

The `execute()`method is also used to execute non-query statements, such as **CREATE**, **ALTER**, **UPDATE**, and **DROP**.

As with queries, the first argument is a string containing one SQL statement.

For most DB packages, `execute()` returns the number of rows affected.

To make changes to the database permanent, changes must be commited with `CONNECTION.commit()`.

Example

db_sqlite_add_row.py

```
from datetime import date
import sqlite3


with sqlite3.connect("../DATA/presidents.db") as s3conn:  # connect to database

    sql_insert = """
    insert into presidents
    (termnum, lastname, firstname, termstart, termend, birthplace, birthstate, birthdate,
deathdate,  party)
    values (47, 'Ramirez', 'Mary', '2025-01-20', null, 'Topeka',
    'Kansas', '1968-09-22', null, 'Independent')
    """

    cursor = s3conn.cursor()

    try:
        cursor.execute(sql_insert)
    except (sqlite3.OperationalError, sqlite3.DatabaseError, sqlite3.DataError) as err:
        print(err)
        s3conn.rollback()
    else:
        s3conn.commit()
    finally:
        cursor.close()
```


Example

db_sqlite_delete_row.py

```
from datetime import date
import sqlite3


with sqlite3.connect("../DATA/presidents.db") as conn:  # connect to DB

    sql_delete = """
    delete from presidents
    where TERMNUM = 47
    """

    cursor = conn.cursor()  # get a cursor

    try:
        cursor.execute(sql_delete)
    except (sqlite3.DatabaseError, sqlite3.OperationalError, sqlite3.DataError) as err:
        print(err)
        conn.rollback()
    else:
        conn.commit()

    cursor.close()
```

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 happens when using string formatting and raw user input to build SQL statements. An attacker can embed malicious SQL commands in input data.

Since the programmer is generating the SQL code as a string, there is no way to check for malicious SQL code. It is best practice to use parameterized statements, which prevents any user input from being *injected* into the SQL statement.

> see http://www.xkcd.com/327 for a well-known web comic on this subject.

Example

db_sql_injection.py

```python
#
good_input = 'Google'
malicious_input = "'; drop table customers; -- "  # input would come from a web form, for instance

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

good_query = naive_format.format(good_input)  # string formatting naively adds the user input to a field, expecting only a customer name
malicious_query = naive_format.format(malicious_input)  # string formatting naively adds the user input to a field, expecting only a customer name

print("Good query:")
print(good_query)  # non-malicious input works fine
print()

print("Bad query:")
print(malicious_query)  # 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

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

All SQL statements may be parameterized, including queries.

Parameterized statements 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.

Table 3. Placeholders for SQL Parameters

Python package	Placeholder
`pymysql`	`%s`
`cx_oracle`	`:param_name`
`pyodbc`	`?`
`pymssql`	`%d` for `int`, `%s` for `str`, etc.
`Psychopg`	`%s` or `%(param_name)s`
`SQLite`	`?` or `:param_name`

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

Example

db_sqlite_parameterized.py

```
import sqlite3

TERMS_TO_UPDATE = [1, 5, 19, 22, 36]

PARTY_UPDATE = '''
update presidents
set party = "SURPRISE!"
where termnum = ?
'''  # ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use different placeholders

PARTY_QUERY = """
select termnum, firstname, lastname, party
from presidents
where termnum = ?
"""

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()

    for termnum in TERMS_TO_UPDATE:
        s3cursor.execute(PARTY_UPDATE, [termnum])  # second argument to execute() is iterable of values to fill in placeholders from left to right

    s3conn.commit()

    for termnum in TERMS_TO_UPDATE:
        s3cursor.execute(PARTY_QUERY, [termnum])
        print(s3cursor.fetchone())
```

db_sqlite_parameterized.py

```
(1, 'George', 'Washington', 'SURPRISE!')
(5, 'James', 'Monroe', 'SURPRISE!')
(19, 'Rutherford Birchard', 'Hayes', 'SURPRISE!')
(22, 'Grover', 'Cleveland', 'SURPRISE!')
(36, 'Lyndon Baines', 'Johnson', 'SURPRISE!')
```

Example

db_sqlite_restore_parties.py

```
import sqlite3

RESTORE_DATA = [
    (1, 'no party'),
    (5, 'Democratic - Republican'),
    (19, 'Republican'),
    (22, 'Democratic'),
    (36, 'Democratic')
]

PARTY_UPDATE = '''
update presidents
set party = ?
where termnum = ?
'''  # ? is SQLite3 placeholder; other DBMSs may use other placeholders

PARTY_QUERY = """
select termnum, firstname, lastname, party
from presidents
where termnum = ?
"""

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()

    for termnum, party in RESTORE_DATA:
        s3cursor.execute(PARTY_UPDATE, [party, termnum])  # second argument to execute()
is iterable of values to fill in placeholders from left to right
    s3conn.commit()

    for termnum, _ in RESTORE_DATA:
        s3cursor.execute(PARTY_QUERY, [termnum])
        print(s3cursor.fetchone())
```

db_sqlite_restore_parties.py

```
(1, 'George', 'Washington', 'no party')
(5, 'James', 'Monroe', 'Democratic - Republican')
(19, 'Rutherford Birchard', 'Hayes', 'Republican')
(22, 'Grover', 'Cleveland', 'Democratic')
(36, 'Lyndon Baines', 'Johnson', 'Democratic')
```


Example

db_sqlite_bulk_insert.py

source

```
import sqlite3
import os
import csv

DATA_FILE = '../DATA/fruit_data.csv'

DB_NAME = 'fruits.db'
DB_TABLE = 'fruits'

SQL_CREATE_TABLE = f"""
create table {DB_TABLE} (
    id integer primary key,
    name varchar(30),
    unit varchar(30),
    unitprice decimal(6, 2)
)
"""  # SQL statement to create table

SQL_INSERT_ROW = f'''
insert into {DB_TABLE} (name, unit, unitprice) values (?, ?, ?)
'''  # parameterized SQL statement to insert one record

SQL_SELECT_ALL = f"""
select name, unit, unitprice from {DB_TABLE}
"""

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

    :return: None
    """
    conn, cursor = get_connection()
    create_database(cursor)
    populate_database(conn, cursor)
    read_database(cursor)

    cursor.close()
    conn.close()
```

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

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)  # remove database if it exists

    conn = sqlite3.connect(DB_NAME)  # connect to (new) database
    cursor = conn.cursor()
    return conn, cursor


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

    :param conn: The database connection
    :return: None
    """
    cursor.execute(SQL_CREATE_TABLE)  # run SQL to create table


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

    :param conn: The database connection
    :return: None
    """
    with open(DATA_FILE) as file_in:
        fruit_data = csv.reader(file_in, quoting=csv.QUOTE_NONNUMERIC)

        for row in fruit_data:
            try:
                # add a row to the table
                cursor.execute(SQL_INSERT_ROW, row)
            except sqlite3.DatabaseError as err:
                print(err)
                conn.rollback()
            else:
                # commit the inserts; without this, no data would be saved
                conn.commit()

def read_database(cursor):
    cursor.execute(SQL_SELECT_ALL)
    for name, unit, unitprice in cursor.fetchall():
```



```
        print(f'{name:12s} {unitprice:5.2f}/{unit}')


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

db_sqlite_bulk_insert.py

```
pomegranate   0.99/each
cherry        2.25/pound
apricot       3.49/pound
date          1.20/pound
apple         0.55/pound
lemon         0.69/each
kiwi          0.88/each
orange        0.49/each
lime          0.49/each
watermelon    4.50/each
guava         2.88/pound
papaya        1.79/pound
fig           2.29/pound
pear          1.10/pound
banana        0.65/pound
```

Example

db_sqlite_execute_many.py

```
import sqlite3
import os
import csv

DATA_FILE = '../DATA/fruit_data.csv'

DB_NAME = 'fruits.db'
DB_TABLE = 'fruits'

SQL_CREATE_TABLE = f"""
create table {DB_TABLE} (
    id integer primary key,
    name varchar(30),
    unit varchar(30),
    unitprice decimal(6, 2)
)
"""  # SQL statement to create table

SQL_INSERT_ROW = f'''
insert into {DB_TABLE} (name, unit, unitprice) values (?, ?, ?)
'''  # parameterized SQL statement to insert one record

SQL_SELECT_ALL = f"""
select name, unit, unitprice from {DB_TABLE}
"""

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

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

    # read database to confirm inserts
    read_database(cursor)

    cursor.close()
    conn.close()
```



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

    :return: SQLite3 connection object.
    """
    if os.path.exists(DB_NAME):
        os.remove(DB_NAME)  # remove existing database if it exists

    conn = sqlite3.connect(DB_NAME)  # connect to (new) database
    cursor = conn.cursor()
    return conn, cursor


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

    :param conn: The database connection
    :return: None
    """
    cursor.execute(SQL_CREATE_TABLE)  # run SQL to create table


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

    :param conn: The database connection
    :return: None
    """
    with open(DATA_FILE) as file_in:
        fruit_data = csv.reader(file_in, quoting=csv.QUOTE_NONNUMERIC)

        try:
            # iterate over rows of input
            # and add each row to database
            cursor.executemany(SQL_INSERT_ROW, fruit_data)
        except sqlite3.DatabaseError as err:
            print(err)
            conn.rollback()
        else:
            # Commit the inserts. Without this, no data
            # would be saved
            conn.commit()

def read_database(cursor):
    cursor.execute(SQL_SELECT_ALL)
```

```python
    for name, unit, unitprice in cursor.fetchall():
        print(f'{name:12s} {unitprice:5.2f}/{unit}')

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

db_sqlite_execute_many.py

```
pomegranate   0.99/each
cherry        2.25/pound
apricot       3.49/pound
date          1.20/pound
apple         0.55/pound
lemon         0.69/each
kiwi          0.88/each
orange        0.49/each
lime          0.49/each
watermelon    4.50/each
guava         2.88/pound
papaya        1.79/pound
fig           2.29/pound
pear          1.10/pound
banana        0.65/pound
```


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.

Not all of the fields will necessarily be populated. For instane, `sqlite3` only provides column names.

Example

db_sqlite_metadata.py

```
"""
    Provide metadata (tables and column names) for a Sqlite3 database
"""
from pprint import pprint
import sqlite3

DB_NAME = "../DATA/presidents.db"
TABLE_QUERY = '''select * from presidents where 1 = 2'''

def main():
    cursor = connect_to_db(DB_NAME)
    show_metadata(cursor)

def connect_to_db(database_file):
    with sqlite3.connect(database_file) as s3conn:
        return s3conn.cursor()

def show_metadata(cursor):
    cursor.execute(TABLE_QUERY)
    pprint(cursor.description)
    print()
    column_names = [column_data[0] for column_data in cursor.description]
    print(f"{column_names = }")

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


db_sqlite_metadata.py

```
(('termnum', None, None, None, None, None, None),
 ('lastname', None, None, None, None, None, None),
 ('firstname', None, None, None, None, None, None),
 ('termstart', None, None, None, None, None, None),
 ('termend', None, None, None, None, None, None),
 ('birthplace', None, None, None, None, None, None),
 ('birthstate', None, None, None, None, None, None),
 ('birthdate', None, None, None, None, None, None),
 ('deathdate', None, None, None, None, None, None),
 ('party', None, None, None, None, None, None))

column_names = ['termnum', 'lastname', 'firstname', 'termstart', 'termend', 'birthplace',
'birthstate', 'birthdate', 'deathdate', 'party']
```

Dictionary Cursors

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

Some DB packages provide dictionary cursors, which return a dictionary for each row, instead of a tuple. The keys are the names of the columns, so columns can be accessed by name rather than position.

Each package that provides a dictionary cursor has its own way of creating a dictionary cursor, although they all work the same way.

The `sqlite3` package provides a `Row` cursor, which can be indexed by position or by column name.

Table 4. Builtin Dictionary Cursors

Package	How to get a dictionary cursor

pymssql	`conn = pymssql.connect (…, as_dict=True)`
	`dcur = conn.cursor()`
	all cursors will be dict cursors

psychopg[1,2]	`import psycopg.extras`
	`conn = psycopg.connect(…)`
	`dcur = conn.cursor(cursor_factory=psycopg.extras.DictCursor)`
	only this cursor will be a dict cursor

sqlite3[1]	`conn = sqlite3.connect (…)`
	`dcur = conn.cursor()`
	`dcur.row_factory = sqlite3.Row`
	only this cursor will be a dict cursor
	
	`conn = sqlite3.connect (…)`
	`conn.row_factory = sqlite3.Row`
	`dcur = conn.cursor()`
	all cursors will be dict cursors

pymysql[1]	`import pymysql.cursors`
	
	`conn = pymysql.connect(…)`
	`dcur = conn.cursor(pymysql.cursors.DictCursor)`
	only this cursor will be a dict cursor
	
	`conn = pymysql.connect(…,`
	`cursorclass = pymysql.cursors.DictCursor`
	`)`
	all cursors will be dict cursors

cx_oracle	*Not available — use db_iterrows*
pyodbc	*Not available — use db_iterrows*
pgdb	*Not available — use db_iterrows*

[1] Cursor supports indexing by either key value (dict style) or integer position (list style), as well as iteration. [2] Also supports `RealDictCursor` which is an actual dictionary, and `NamedTupleCursor`, which is an actual `namedtuple`

Example

db_sqlite_dict_cursor.py

```
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)

dict_cursor = s3conn.cursor()  # get a normal SQLite3 cursor

# make _this_ cursor a dictionary cursor
dict_cursor.row_factory = sqlite3.Row  # set the row factory to be a Row object

# Row objects are dict/list hybrids -- row[name] or row[pos]

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

for row in dict_cursor.fetchall():
    print(row['firstname'], row['lastname'])  # index row by column name

print('-' * 50)
```

db_sqlite_dict_cursor.py

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

Generic alternate cursors

- Create generator function
  - Get column names from *cursor*.description()
  - For each row
    - Make object from column names and values
      - Dictionary
      - Named tuple
      - Dataclass

For database modules that don't provice a dictionary cursor, the `iterrows_asdict()` function described below can be used with a cursor from any DB API-compliant package.

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. `db_iterrows` also provides `iterrows_asnamedtuple()`, which returns each row as a named tuple, and `iterrows_asdataclass()`, which returns each row as an instance of a custom dataclass.

The functions in `db_iterrows` return generator objects. When you loop over the generator object, each element is a dictionary, named tuple, or instance of a dataclass, depending on which function you called.

Example

db_iterrows.py

```python
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
from collections import namedtuple
from dataclasses import make_dataclass

def get_column_names(cursor):
    return [desc[0] for desc in cursor.description]

def iterrows_asdict(cursor):
    '''Generate rows as dictionaries'''
    column_names = get_column_names(cursor)
    for row in cursor.fetchall():
        row_dict = dict(zip(column_names, row))
        yield row_dict

def iterrows_asnamedtuple(cursor):
    '''Generate rows as named tuples'''
    column_names = get_column_names(cursor)
    Row = namedtuple('Row', column_names)
    for row in cursor.fetchall():
        yield Row(*row)

def iterrows_asdataclass(cursor):
    '''Generate rows as dataclass instances'''
    column_names = get_column_names(cursor)
    Row = make_dataclass('row_tuple', column_names)

    for row in cursor.fetchall():
        yield Row(*row)
```

Example

db_sqlite_iterrows.py

```
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
import sqlite3
from db_iterrows import *

sql_select = """
SELECT firstname, lastname, party
FROM presidents
WHERE termnum > 39
"""

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

cursor = conn.cursor()

cursor.execute(sql_select)

for row in iterrows_asdict(cursor):
    print(row['firstname'], row['lastname'], row['party'])

print('-' * 60)


cursor.execute(sql_select)

for row in iterrows_asnamedtuple(cursor):
    print(row.firstname, row.lastname, row.party)

print('-' * 60)


cursor.execute(sql_select)

for row in iterrows_asdataclass(cursor):
    print(row.firstname, row.lastname, row.party)
```


db_sqlite_iterrows.py

```
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
------------------------------------------------------------
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
------------------------------------------------------------
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
```

Transactions

- Transactions allow safer control of updates
- `commit()` to make database changes permanent
- `rollback()` to discard changes

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. If you don't call `commit()` after modifying a table, the data will not be saved.

You can also turn on *autocommit*, which calls `commit()` after every statement. See the table below for how autocommit is implemented in various DB packages. This is not considered a best practice.

Table 5. How to turn on autocommit

Package	Method/Attribute
cx_oracle	`conn.autocommit = True`
ibm_db_api	`conn.set_autocommit(True)`

pymysql	`pymysql.connect(…, autocommit=True)`
	or
	`conn.autocommit(True)`

psycopg	`conn.autocommit = True`
sqlite3	`sqlite3.connect(dbname, isolation_level=None)`

pymysql only supports transaction processing when using the **InnoDB** engine

Example

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

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.

Chapter 1 Exercises

Exercise 1-1 (president_sqlite.py, president_main_sqlite.py)

Part A (president_sqlite.py)

For this exercise, use the SQLite database named `presidents.db` in the DATA folder. It has the following layout

Table 6. Layout of President Table

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

Refactor the `president.py` module to get its data from this table, rather than from a file.

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 lab files.

Part B (president_main_sqlite.py)

Modify `president_main.py` that used `president.py`. It should import the `President` class from this new module that uses the database instead of a text file, but otherwise work the same as before.

Exercise 1-2 (add_pres_sqlite.py)

Add another president to the presidents database. Just make up the data for your new president.

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 (?,?,...)
```

Index

T