# sqlite3 — Embedded Relational Database

**Purpose:** Implements an embedded relational database with SQL support.

The `sqlite3` module implements a Python DB-API 2.0 compliant interface to SQLite, an in-process relational database. SQLite is designed to be embedded in applications, instead of using a separate database server program such as MySQL, PostgreSQL, or Oracle. It is fast, rigorously tested, and flexible, making it suitable for prototyping and production deployment for some applications.

## Creating a Database

An SQLite database is stored as a single file on the file system. The library manages access to the file, including locking it to prevent corruption when multiple writers use it. The database is created the first time the file is accessed, but the application is responsible for managing the table definitions, or *schema*, within the database.

This example looks for the database file before opening it with `connect()` so it knows when to create the schema for new databases.

```
# sqlite3_createdb.py

import os
import sqlite3

db_filename = 'todo.db'

db_is_new = not os.path.exists(db_filename)

conn = sqlite3.connect(db_filename)

if db_is_new:
    print('Need to create schema')
else:
    print('Database exists, assume schema does, too.')

conn.close()
```

Running the script twice shows that it creates the empty file if it does not exist.

```
$ ls *.db

ls: *.db: No such file or directory

$ python3 sqlite3_createdb.py

Need to create schema

$ ls *.db

todo.db

$ python3 sqlite3_createdb.py

Database exists, assume schema does, too.
```

After creating the new database file, the next step is to create the schema to define the tables within the database. The remaining examples in this section all use the same database schema with tables for managing tasks. The details of the database schema are presented in the table below and the table below.

The project Table

| Column | Type | Description |
|---|---|---|
| name | text | Project name |
| description | text | Long project description |

| | | |
|---|---|---|
| deadline | date | Due date for the entire project |

The task Table

| Column | Type | Description |
|---|---|---|
| id | number | Unique task identifier |
| priority | integer | Numerical priority, lower is more important |
| details | text | Full task details |
| status | text | Task status (one of ‘new’, ‘pending’, ‘done’, or ‘canceled’). |
| deadline | date | Due date for this task |
| completed_on | date | When the task was completed. |
| project | text | The name of the project for this task. |

The *data definition language* (DDL) statements to create the tables are:

```
-- todo_schema.sql

-- Schema for to-do application examples.

-- Projects are high-level activities made up of tasks
create table project (
    name        text primary key,
    description text,
    deadline    date
);

-- Tasks are steps that can be taken to complete a project
create table task (
    id           integer primary key autoincrement not null,
    priority     integer default 1,
    details      text,
    status       text,
    deadline     date,
    completed_on date,
    project      text not null references project(name)
);
```

The `executescript()` method of the `Connection` can be used to run the DDL instructions to create the schema.

```
# sqlite3_create_schema.py

import os
import sqlite3

db_filename = 'todo.db'
schema_filename = 'todo_schema.sql'

db_is_new = not os.path.exists(db_filename)

with sqlite3.connect(db_filename) as conn:
    if db_is_new:
        print('Creating schema')
        with open(schema_filename, 'rt') as f:
            schema = f.read()
        conn.executescript(schema)

        print('Inserting initial data')

        conn.executescript("""
        insert into project (name, description, deadline)
        values ('pymotw', 'Python Module of the Week',
                '2016-11-01');

        insert into task (details, status, deadline, project)
        values ('write about select', 'done', '2016-04-25',
```

```
                'pymotw');

        insert into task (details, status, deadline, project)
        values ('write about random', 'waiting', '2016-08-22',
                'pymotw');

        insert into task (details, status, deadline, project)
        values ('write about sqlite3', 'active', '2017-07-31',
                'pymotw');
        """)
    else:
        print('Database exists, assume schema does, too.')
```

After the tables are created, a few `insert` statements create a sample project and related tasks. The `sqlite3` command line program can be used to examine the contents of the database.

```
$ rm -f todo.db
$ python3 sqlite3_create_schema.py

Creating schema
Inserting initial data

$ sqlite3 todo.db 'select * from task'

1|1|write about select|done|2016-04-25||pymotw
2|1|write about random|waiting|2016-08-22||pymotw
3|1|write about sqlite3|active|2017-07-31||pymotw
```

## Retrieving Data

To retrieve the values saved in the `task` table from within a Python program, create a `Cursor` from a database connection. A cursor produces a consistent view of the data, and is the primary means of interacting with a transactional database system like SQLite.

```
# sqlite3_select_tasks.py

import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
    select id, priority, details, status, deadline from task
    where project = 'pymotw'
    """)

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

Querying is a two step process. First, run the query with the cursor's `execute()` method to tell the database engine what data to collect. Then, use `fetchall()` to retrieve the results. The return value is a sequence of tuples containing the values for the columns included in the `select` clause of the query.

```
$ python3 sqlite3_select_tasks.py

 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

The results can be retrieved one at a time with `fetchone()`, or in fixed-size batches with `fetchmany()`.

```
# sqlite3_select_variations.py

import sqlite3
```

```
db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
    select name, description, deadline from project
    where name = 'pymotw'
    """)
    name, description, deadline = cursor.fetchone()

    print('Project details for {} ({})\n  due {}'.format(
        description, name, deadline))

    cursor.execute("""
    select id, priority, details, status, deadline from task
    where project = 'pymotw' order by deadline
    """)

    print('\nNext 5 tasks:')
    for row in cursor.fetchmany(5):
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

The value passed to `fetchmany()` is the maximum number of items to return. If fewer items are available, the sequence returned will be smaller than the maximum value.

```
$ python3 sqlite3_select_variations.py

Project details for Python Module of the Week (pymotw)
  due 2016-11-01

Next 5 tasks:
 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

## Query Metadata

The DB-API 2.0 specification says that after `execute()` has been called, the `Cursor` should set its `description` attribute to hold information about the data that will be returned by the fetch methods. The API specification say that the description value is a sequence of tuples containing the column name, type, display size, internal size, precision, scale, and a flag that says whether null values are accepted.

```
# sqlite3_cursor_description.py

import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    cursor.execute("""
    select * from task where project = 'pymotw'
    """)

    print('Task table has these columns:')
    for colinfo in cursor.description:
        print(colinfo)
```

Because `sqlite3` does not enforce type or size constraints on data inserted into a database, only the column name value is filled in.

```
$ python3 sqlite3_cursor_description.py

Task table has these columns:
('id', None, None, None, None, None, None)
```

```
('id', None, None, None, None, None, None)
('priority', None, None, None, None, None, None)
('details', None, None, None, None, None, None)
('status', None, None, None, None, None, None)
('deadline', None, None, None, None, None, None)
('completed_on', None, None, None, None, None, None)
('project', None, None, None, None, None, None)
```

## Row Objects

By default, the values returned by the fetch methods as "rows" from the database are tuples. The caller is responsible for knowing the order of the columns in the query and extracting individual values from the tuple. When the number of values in a query grows, or the code working with the data is spread out in a library, it is usually easier to work with an object and access values using their column names. That way, the number and order of the tuple contents can change over time as the query is edited, and code depending on the query results is less likely to break.

`Connection` objects have a `row_factory` property that allows the calling code to control the type of object created to represent each row in the query result set. `sqlite3` also includes a `Row` class intended to be used as a row factory. Column values can be accessed through `Row` instances by using the column index or name.

```
# sqlite3_row_factory.py

import sqlite3

db_filename = 'todo.db'

with sqlite3.connect(db_filename) as conn:
    # Change the row factory to use Row
    conn.row_factory = sqlite3.Row

    cursor = conn.cursor()

    cursor.execute("""
    select name, description, deadline from project
    where name = 'pymotw'
    """)
    name, description, deadline = cursor.fetchone()

    print('Project details for {} ({})\n  due {}'.format(
        description, name, deadline))

    cursor.execute("""
    select id, priority, status, deadline, details from task
    where project = 'pymotw' order by deadline
    """)

    print('\nNext 5 tasks:')
    for row in cursor.fetchmany(5):
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            row['id'], row['priority'], row['details'],
            row['status'], row['deadline'],
        ))
```

This version of the `sqlite3_select_variations.py` example has been re-written using `Row` instances instead of tuples. The row from the project table is still printed by accessing the column values through position, but the `print` statement for tasks uses keyword lookup instead, so it does not matter that the order of the columns in the query has been changed.

```
$ python3 sqlite3_row_factory.py

Project details for Python Module of the Week (pymotw)
  due 2016-11-01

Next 5 tasks:
 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

## Using Variables with Queries

Using queries defined as literal strings embedded in a program is inflexible. For example, when another project is added to the database the query to show the top five tasks should be updated to work with either project. One way to add more flexibility is to build an SQL statement with the desired query by combining values in Python. However, building a query string in this way is dangerous, and should be avoided. Failing to correctly escape special characters in the variable parts of the query can result in SQL parsing errors, or worse, a class of security vulnerabilities known as *SQL-injection attacks*, which allow intruders to execute arbitrary SQL statements in the database.

The proper way to use dynamic values with queries is through *host variables* passed to `execute()` along with the SQL instruction. A placeholder value in the SQL is replaced with the value of the host variable when the statement is executed. Using host variables instead of inserting arbitrary values into the SQL before it is parsed avoids injection attacks because there is no chance that the untrusted values will affect how the SQL is parsed. SQLite supports two forms for queries with placeholders, positional and named.

## Positional Parameters

A question mark (?) denotes a positional argument, passed to `execute()` as a member of a tuple.

```
# sqlite3_argument_positional.py

import sqlite3
import sys

db_filename = 'todo.db'
project_name = sys.argv[1]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    query = """
    select id, priority, details, status, deadline from task
    where project = ?
    """

    cursor.execute(query, (project_name,))

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

The command line argument is passed safely to the query as a positional argument, and there is no chance for bad data to corrupt the database.

```
$ python3 sqlite3_argument_positional.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

## Named Parameters

Use named parameters for more complex queries with a lot of parameters, or where some parameters are repeated multiple times within the query. Named parameters are prefixed with a colon (e.g., `:param_name`).

```
# sqlite3_argument_named.py

import sqlite3
import sys

db_filename = 'todo.db'
project_name = sys.argv[1]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()

    query = """
    select id, priority, details, status, deadline from task
    where project = :project_name
    order by deadline, priority
    """
```

```
    """

    cursor.execute(query, {'project_name': project_name})

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

Neither positional nor named parameters need to be quoted or escaped, since they are given special treatment by the query parser.

```
$ python3 sqlite3_argument_named.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [waiting ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

Query parameters can be used with `select`, `insert`, and `update` statements. They can appear in any part of the query where a literal value is legal.

```
# sqlite3_argument_update.py

import sqlite3
import sys

db_filename = 'todo.db'
id = int(sys.argv[1])
status = sys.argv[2]

with sqlite3.connect(db_filename) as conn:
    cursor = conn.cursor()
    query = "update task set status = :status where id = :id"
    cursor.execute(query, {'status': status, 'id': id})
```

This `update` statement uses two named parameters. The `id` value is used to find the right row to modify, and the `status` value is written to the table.

```
$ python3 sqlite3_argument_update.py 2 done
$ python3 sqlite3_argument_named.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 2 [1] write about random        [done    ] (2016-08-22)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

## Bulk Loading

To apply the same SQL instruction to a large set of data, use `executemany()`. This is useful for loading data, since it avoids looping over the inputs in Python and lets the underlying library apply loop optimizations. This example program reads a list of tasks from a comma-separated value file using the `csv` module and loads them into the database.

```
# sqlite3_load_csv.py

import csv
import sqlite3
import sys

db_filename = 'todo.db'
data_filename = sys.argv[1]

SQL = """
insert into task (details, priority, status, deadline, project)
values (:details, :priority, 'active', :deadline, :project)
"""

with open(data_filename, 'rt') as csv_file:
    csv_reader = csv.DictReader(csv_file)

    with sqlite3.connect(db_filename) as conn:
```

```
    with sqlite3.connect(db_filename) as conn:
        cursor = conn.cursor()
        cursor.executemany(SQL, csv_reader)
```

The sample data file `tasks.csv` contains:

```
deadline,project,priority,details
2016-11-30,pymotw,2,"finish reviewing markup"
2016-08-20,pymotw,2,"revise chapter intros"
2016-11-01,pymotw,1,"subtitle"
```

Running the program produces:

```
$ python3 sqlite3_load_csv.py tasks.csv
$ python3 sqlite3_argument_named.py pymotw

 1 [1] write about select        [done    ] (2016-04-25)
 5 [2] revise chapter intros     [active  ] (2016-08-20)
 2 [1] write about random        [done    ] (2016-08-22)
 6 [1] subtitle                  [active  ] (2016-11-01)
 4 [2] finish reviewing markup   [active  ] (2016-11-30)
 3 [1] write about sqlite3       [active  ] (2017-07-31)
```

## Defining New Column Types

SQLite has native support for integer, floating point, and text columns. Data of these types is converted automatically by `sqlite3` from Python's representation to a value that can be stored in the database, and back again, as needed. Integer values are loaded from the database into `int` or `long` variables, depending on the size of the value. Text is saved and retrieved as `str`, unless the `text_factory` for the `Connection` has been changed.

Although SQLite only supports a few data types internally, `sqlite3` includes facilities for defining custom types to allow a Python application to store any type of data in a column. Conversion for types beyond those supported by default is enabled in the database connection using the `detect_types` flag. Use `PARSE_DECLTYPES` if the column was declared using the desired type when the table was defined.

```
# sqlite3_date_types.py

import sqlite3
import sys

db_filename = 'todo.db'

sql = "select id, details, deadline from task"


def show_deadline(conn):
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()
    cursor.execute(sql)
    row = cursor.fetchone()
    for col in ['id', 'details', 'deadline']:
        print('  {:<8}  {!r:<26} {}'.format(
            col, row[col], type(row[col])))
    return


print('Without type detection:')
with sqlite3.connect(db_filename) as conn:
    show_deadline(conn)

print('\nWith type detection:')
with sqlite3.connect(db_filename,
                     detect_types=sqlite3.PARSE_DECLTYPES,
                     ) as conn:
    show_deadline(conn)
```

`sqlite3` provides converters for date and timestamp columns, using the classes `date` and `datetime` from the `datetime` module to represent the values in Python. Both date-related converters are enabled automatically when type-detection is turned on.

```
$ python3 sqlite3_date_types.py

Without type detection:
  id        1                          <class 'int'>
  details   'write about select'       <class 'str'>
  deadline  '2016-04-25'               <class 'str'>

With type detection:
  id        1                          <class 'int'>
  details   'write about select'       <class 'str'>
  deadline  datetime.date(2016, 4, 25) <class 'datetime.date'>
```

Two functions need to be registered to define a new type. The *adapter* takes the Python object as input and returns a byte string that can be stored in the database. The *converter* receives the string from the database and returns a Python object. Use `register_adapter()` to define an adapter function, and `register_converter()` for a converter function.

```
# sqlite3_custom_type.py
```

```
import pickle
import sqlite3

db_filename = 'todo.db'


def adapter_func(obj):
    """Convert from in-memory to storage representation.
    """
    print('adapter_func({})\n'.format(obj))
    return pickle.dumps(obj)


def converter_func(data):
    """Convert from storage to in-memory representation.
    """
    print('converter_func({!r})\n'.format(data))
    return pickle.loads(data)


class MyObj:

    def __init__(self, arg):
        self.arg = arg

    def __str__(self):
        return 'MyObj({!r})'.format(self.arg)


# Register the functions for manipulating the type.
sqlite3.register_adapter(MyObj, adapter_func)
sqlite3.register_converter("MyObj", converter_func)

# Create some objects to save.  Use a list of tuples so
# the sequence can be passed directly to executemany().
to_save = [
    (MyObj('this is a value to save'),),
    (MyObj(42),),
]

with sqlite3.connect(
        db_filename,
        detect_types=sqlite3.PARSE_DECLTYPES) as conn:
    # Create a table with column of type "MyObj"
    conn.execute("""
    create table if not exists obj (
        id    integer primary key autoincrement not null,
        data  MyObj
    )
    """)
    cursor = conn.cursor()
```

```
    # Insert the objects into the database
    cursor.executemany("insert into obj (data) values (?)",
                       to_save)

    # Query the database for the objects just saved
    cursor.execute("select id, data from obj")
    for obj_id, obj in cursor.fetchall():
        print('Retrieved', obj_id, obj)
        print('  with type', type(obj))
        print()
```

This example uses pickle to save an object to a string that can be stored in the database, a useful technique for storing arbitrary objects, but one that does not allow querying based on object attributes. A real *object-relational mapper*, such as SQLAlchemy, that stores attribute values in their own columns will be more useful for large amounts of data.

```
$ python3 sqlite3_custom_type.py

adapter_func(MyObj('this is a value to save'))

adapter_func(MyObj(42))

converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\x0
3\x00\x00\x00argq\x03X\x17\x00\x00\x00this is a value to saveq\x04sb
.')

converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\x0
3\x00\x00\x00argq\x03K*sb.')

Retrieved 1 MyObj('this is a value to save')
  with type <class '__main__.MyObj'>

Retrieved 2 MyObj(42)
  with type <class '__main__.MyObj'>
```

## Determining Types for Columns

There are two sources for types information about the data for a query. The original table declaration can be used to identify the type of a real column, as shown earlier. A type specifier can also be included in the `select` clause of the query itself using the form as `"name [type]"`.

```
# sqlite3_custom_type_column.py

import pickle
import sqlite3

db_filename = 'todo.db'


def adapter_func(obj):
    """Convert from in-memory to storage representation.
    """
    print('adapter_func({})\n'.format(obj))
    return pickle.dumps(obj)


def converter_func(data):
    """Convert from storage to in-memory representation.
    """
    print('converter_func({!r})\n'.format(data))
    return pickle.loads(data)


class MyObj:

    def __init__(self, arg):
        self.arg = arg

    def __str__(self):
        return 'MyObj({!r})'.format(self.arg)
```

```
# Register the functions for manipulating the type.
sqlite3.register_adapter(MyObj, adapter_func)
sqlite3.register_converter("MyObj", converter_func)

# Create some objects to save.  Use a list of tuples so we
# can pass this sequence directly to executemany().
to_save = [
    (MyObj('this is a value to save'),),
    (MyObj(42),),
]

with sqlite3.connect(
        db_filename,
        detect_types=sqlite3.PARSE_COLNAMES) as conn:
    # Create a table with column of type "text"
    conn.execute("""
    create table if not exists obj2 (
        id    integer primary key autoincrement not null,
        data  text
    )
    """)
    cursor = conn.cursor()

    # Insert the objects into the database
    cursor.executemany("insert into obj2 (data) values (?)",
                       to_save)

    # Query the database for the objects just saved,
    # using a type specifier to convert the text
    # to objects.
    cursor.execute(
        'select id, data as "pickle [MyObj]" from obj2',
    )
    for obj_id, obj in cursor.fetchall():
        print('Retrieved', obj_id, obj)
        print('  with type', type(obj))
        print()
```

Use the `detect_types` flag `PARSE_COLNAMES` when the type is part of the query instead of the original table definition.

```
$ python3 sqlite3_custom_type_column.py

adapter_func(MyObj('this is a value to save'))

adapter_func(MyObj(42))

converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\x0
3\x00\x00\x00argq\x03X\x17\x00\x00\x00this is a value to saveq\x04sb
.')

converter_func(b'\x80\x03c__main__\nMyObj\nq\x00)\x81q\x01}q\x02X\x0
3\x00\x00\x00argq\x03K*sb.')

Retrieved 1 MyObj('this is a value to save')
  with type <class '__main__.MyObj'>

Retrieved 2 MyObj(42)
  with type <class '__main__.MyObj'>
```

# Transactions

One of the key features of relational databases is the use of *transactions* to maintain a consistent internal state. With transactions enabled, several changes can be made through one connection without effecting any other users until the results are *committed* and flushed to the actual database.

## Preserving Changes

Changes to the database, either through `insert` or `update` statements, need to be saved by explicitly calling `commit()`. This

requirement gives an application an opportunity to make several related changes together, so they are stored *atomically* instead of incrementally, and avoids a situation where partial updates are seen by different clients connecting to the database simultaneously.

The effect of calling `commit()` can be seen with a program that uses several connections to the database. A new row is inserted with the first connection, and then two attempts are made to read it back using separate connections.

```
# sqlite3_transaction_commit.py

import sqlite3

db_filename = 'todo.db'


def show_projects(conn):
    cursor = conn.cursor()
    cursor.execute('select name, description from project')
    for name, desc in cursor.fetchall():
        print('  ', name)


with sqlite3.connect(db_filename) as conn1:
    print('Before changes:')
    show_projects(conn1)

    # Insert in one cursor
    cursor1 = conn1.cursor()
    cursor1.execute("""
    insert into project (name, description, deadline)
    values ('virtualenvwrapper', 'Virtualenv Extensions',
            '2011-01-01')
    """)

    print('\nAfter changes in conn1:')
    show_projects(conn1)

    # Select from another connection, without committing first
    print('\nBefore commit:')
    with sqlite3.connect(db_filename) as conn2:
        show_projects(conn2)

    # Commit then select from another connection
    conn1.commit()
    print('\nAfter commit:')
    with sqlite3.connect(db_filename) as conn3:
        show_projects(conn3)
```

When `show_projects()` is called before `conn1` has been committed, the results depend on which connection is used. Since the change was made through `conn1`, it sees the altered data. However, `conn2` does not. After committing, the new connection `conn3` sees the inserted row.

```
$ python3 sqlite3_transaction_commit.py

Before changes:
   pymotw

After changes in conn1:
   pymotw
   virtualenvwrapper

Before commit:
   pymotw

After commit:
   pymotw
   virtualenvwrapper
```

## Discarding Changes

Uncommitted changes can also be discarded entirely using `rollback()`. The `commit()` and `rollback()` methods are usually

called from different parts of the same `try:except` block, with errors triggering a rollback.

```
# sqlite3_transaction_rollback.py

import sqlite3

db_filename = 'todo.db'


def show_projects(conn):
    cursor = conn.cursor()
    cursor.execute('select name, description from project')
    for name, desc in cursor.fetchall():
        print('  ', name)


with sqlite3.connect(db_filename) as conn:

    print('Before changes:')
    show_projects(conn)

    try:

        # Insert
        cursor = conn.cursor()
        cursor.execute("""delete from project
                       where name = 'virtualenvwrapper'
                       """)

        # Show the settings
        print('\nAfter delete:')
        show_projects(conn)

        # Pretend the processing caused an error
        raise RuntimeError('simulated error')

    except Exception as err:
        # Discard the changes
        print('ERROR:', err)
        conn.rollback()

    else:
        # Save the changes
        conn.commit()

    # Show the results
    print('\nAfter rollback:')
    show_projects(conn)
```

After calling `rollback()`, the changes to the database are no longer present.

```
$ python3 sqlite3_transaction_rollback.py

Before changes:
   pymotw
   virtualenvwrapper

After delete:
   pymotw
ERROR: simulated error

After rollback:
   pymotw
   virtualenvwrapper
```

## Isolation Levels

`sqlite3` supports three locking modes, called *isolation levels*, that control the technique used to prevent incompatible changes between connections. The isolation level is set by passing a string as the `isolation_level` argument when a connection is opened, so different connections can use different values.

This program demonstrates the effect of different isolation levels on the order of events in threads using separate connections to the same database. Four threads are created. Two threads write changes to the database by updating existing rows. The other two threads attempt to read all of the rows from the `task` table.

```
# sqlite3_isolation_levels.py

import logging
import sqlite3
import sys
import threading
import time

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s (%(threadName)-10s) %(message)s',
)

db_filename = 'todo.db'
isolation_level = sys.argv[1]


def writer():
    with sqlite3.connect(
            db_filename,
            isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        cursor.execute('update task set priority = priority + 1')
        logging.debug('waiting to synchronize')
        ready.wait()  # synchronize threads
        logging.debug('PAUSING')
        time.sleep(1)
        conn.commit()
        logging.debug('CHANGES COMMITTED')


def reader():
    with sqlite3.connect(
            db_filename,
            isolation_level=isolation_level) as conn:
        cursor = conn.cursor()
        logging.debug('waiting to synchronize')
        ready.wait()  # synchronize threads
        logging.debug('wait over')
        cursor.execute('select * from task')
        logging.debug('SELECT EXECUTED')
        cursor.fetchall()
        logging.debug('results fetched')


if __name__ == '__main__':
    ready = threading.Event()

    threads = [
        threading.Thread(name='Reader 1', target=reader),
        threading.Thread(name='Reader 2', target=reader),
        threading.Thread(name='Writer 1', target=writer),
        threading.Thread(name='Writer 2', target=writer),
    ]

    [t.start() for t in threads]

    time.sleep(1)
    logging.debug('setting ready')
    ready.set()

    [t.join() for t in threads]
```

The threads are synchronized using an Event object from the threading module. The `writer()` function connects and make changes to the database, but does not commit before the event fires. The `reader()` function connects, then waits to query the database until after the synchronization event occurs.

database until after the synchronization event occurs.

## Deferred

The default isolation level is DEFERRED. Using deferred mode locks the database, but only once a change is begun. All of the previous examples use deferred mode.

```
$ python3 sqlite3_isolation_levels.py DEFERRED

2016-08-20 17:46:26,972 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:26,972 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:26,973 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:27,977 (MainThread) setting ready
2016-08-20 17:46:27,979 (Reader 1  ) wait over
2016-08-20 17:46:27,979 (Writer 1  ) PAUSING
2016-08-20 17:46:27,979 (Reader 2  ) wait over
2016-08-20 17:46:27,981 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:27,982 (Reader 1  ) results fetched
2016-08-20 17:46:27,982 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:27,982 (Reader 2  ) results fetched
2016-08-20 17:46:28,985 (Writer 1  ) CHANGES COMMITTED
2016-08-20 17:46:29,043 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:29,043 (Writer 2  ) PAUSING
2016-08-20 17:46:30,044 (Writer 2  ) CHANGES COMMITTED
```

## Immediate

Immediate mode locks the database as soon as a change starts and prevents other cursors from making changes until the transaction is committed. It is suitable for a database with complicated writes, but more readers than writers, since the readers are not blocked while the transaction is ongoing.

```
$ python3 sqlite3_isolation_levels.py IMMEDIATE

2016-08-20 17:46:30,121 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:30,121 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:30,123 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:31,122 (MainThread) setting ready
2016-08-20 17:46:31,122 (Reader 1  ) wait over
2016-08-20 17:46:31,122 (Reader 2  ) wait over
2016-08-20 17:46:31,122 (Writer 1  ) PAUSING
2016-08-20 17:46:31,124 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:31,124 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:31,125 (Reader 2  ) results fetched
2016-08-20 17:46:31,125 (Reader 1  ) results fetched
2016-08-20 17:46:32,128 (Writer 1  ) CHANGES COMMITTED
2016-08-20 17:46:32,199 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:32,199 (Writer 2  ) PAUSING
2016-08-20 17:46:33,200 (Writer 2  ) CHANGES COMMITTED
```

## Exclusive

Exclusive mode locks the database to all readers and writers. Its use should be limited in situations where database performance is important, since each exclusive connection blocks all other users.

```
$ python3 sqlite3_isolation_levels.py EXCLUSIVE

2016-08-20 17:46:33,320 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:33,320 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:33,324 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:34,323 (MainThread) setting ready
2016-08-20 17:46:34,323 (Reader 1  ) wait over
2016-08-20 17:46:34,323 (Writer 2  ) PAUSING
2016-08-20 17:46:34,323 (Reader 2  ) wait over
2016-08-20 17:46:35,327 (Writer 2  ) CHANGES COMMITTED
2016-08-20 17:46:35,368 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:35,368 (Reader 2  ) results fetched
2016-08-20 17:46:35,369 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:35,369 (Reader 1  ) results fetched
2016-08-20 17:46:35,385 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:35,385 (Writer 1  ) PAUSING
```

```
2016-08-20 17:46:36,386 (Writer 1  ) CHANGES COMMITTED
```

Because the first writer has started making changes, the readers and second writer block until it commits. The `sleep()` call introduces an artificial delay in the writer thread to highlight the fact that the other connections are blocking.

### Autocommit

The `isolation_level` parameter for the connection can also be set to `None` to enable autocommit mode. With autocommit enabled, each `execute()` call is committed immediately when the statement finishes. Autocommit mode is suited for short transactions, such as those that insert a small amount of data into a single table. The database is locked for as little time as possible, so there is less chance of contention between threads.

In `sqlite3_autocommit.py`, the explicit call to `commit()` has been removed and the isolation level is set to `None`, but otherwise is the same as `sqlite3_isolation_levels.py`. The output is different, however, since both writer threads finish their work before either reader starts querying.

```
$ python3 sqlite3_autocommit.py

2016-08-20 17:46:36,451 (Reader 1  ) waiting to synchronize
2016-08-20 17:46:36,451 (Reader 2  ) waiting to synchronize
2016-08-20 17:46:36,455 (Writer 1  ) waiting to synchronize
2016-08-20 17:46:36,456 (Writer 2  ) waiting to synchronize
2016-08-20 17:46:37,452 (MainThread) setting ready
2016-08-20 17:46:37,452 (Reader 1  ) wait over
2016-08-20 17:46:37,452 (Writer 2  ) PAUSING
2016-08-20 17:46:37,452 (Reader 2  ) wait over
2016-08-20 17:46:37,453 (Writer 1  ) PAUSING
2016-08-20 17:46:37,453 (Reader 1  ) SELECT EXECUTED
2016-08-20 17:46:37,454 (Reader 2  ) SELECT EXECUTED
2016-08-20 17:46:37,454 (Reader 1  ) results fetched
2016-08-20 17:46:37,454 (Reader 2  ) results fetched
```

## In-Memory Databases

SQLite supports managing an entire database in RAM, instead of relying on a disk file. In-memory databases are useful for automated testing, where the database does not need to be preserved between test runs, or when experimenting with a schema or other database features. To open an in-memory database, use the string `':memory:'` instead of a filename when creating the `Connection`. Each `':memory:'` connection creates a separate database instance, so changes made by a cursor in one do not effect other connections.

## Exporting the Contents of a Database

The contents of an in-memory database can be saved using the `iterdump()` method of the `Connection`. The iterator returned by `iterdump()` produces a series of strings that together build SQL instructions to recreate the state of the database.

```
# sqlite3_iterdump.py

import sqlite3

schema_filename = 'todo_schema.sql'

with sqlite3.connect(':memory:') as conn:
    conn.row_factory = sqlite3.Row

    print('Creating schema')
    with open(schema_filename, 'rt') as f:
        schema = f.read()
    conn.executescript(schema)

    print('Inserting initial data')
    conn.execute("""
    insert into project (name, description, deadline)
    values ('pymotw', 'Python Module of the Week',
            '2010-11-01')
    """)
    data = [
        ('write about select', 'done', '2010-10-03',
         'pymotw'),
        ('write about random', 'waiting', '2010-10-10',
```

```
         'pymotw'),
        ('write about sqlite3', 'active', '2010-10-17',
         'pymotw'),
    ]
    conn.executemany("""
    insert into task (details, status, deadline, project)
    values (?, ?, ?, ?)
    """, data)

    print('Dumping:')
    for text in conn.iterdump():
        print(text)
```

`iterdump()` can also be used with databases saved to files, but it is most useful for preserving a database that would not otherwise be saved. This output has been edited to fit on the page while remaining syntactically correct.

```
$ python3 sqlite3_iterdump.py

Creating schema
Inserting initial data
Dumping:
BEGIN TRANSACTION;
CREATE TABLE project (
    name        text primary key,
    description text,
    deadline    date
);
INSERT INTO "project" VALUES('pymotw','Python Module of the
Week','2010-11-01');
DELETE FROM "sqlite_sequence";
INSERT INTO "sqlite_sequence" VALUES('task',3);
CREATE TABLE task (
    id           integer primary key autoincrement not null,
    priority     integer default 1,
    details      text,
    status       text,
    deadline     date,
    completed_on date,
    project      text not null references project(name)
);
INSERT INTO "task" VALUES(1,1,'write about
select','done','2010-10-03',NULL,'pymotw');
INSERT INTO "task" VALUES(2,1,'write about
random','waiting','2010-10-10',NULL,'pymotw');
INSERT INTO "task" VALUES(3,1,'write about
sqlite3','active','2010-10-17',NULL,'pymotw');
COMMIT;
```

## Using Python Functions in SQL

SQL syntax supports calling functions during queries, either in the column list or `where` clause of the `select` statement. This feature makes it possible to process data before returning it from the query, and can be used to convert between different formats, perform calculations that would be clumsy in pure SQL, and reuse application code.

```
# sqlite3_create_function.py

import codecs
import sqlite3

db_filename = 'todo.db'


def encrypt(s):
    print('Encrypting {!r}'.format(s))
    return codecs.encode(s, 'rot-13')


def decrypt(s):
    print('Decrypting {!r}'.format(s))
    return codecs.encode(s, 'rot-13')
```

```
    return codecs.encode(s, 'rot-13')


with sqlite3.connect(db_filename) as conn:

    conn.create_function('encrypt', 1, encrypt)
    conn.create_function('decrypt', 1, decrypt)
    cursor = conn.cursor()

    # Raw values
    print('Original values:')
    query = "select id, details from task"
    cursor.execute(query)
    for row in cursor.fetchall():
        print(row)

    print('\nEncrypting...')
    query = "update task set details = encrypt(details)"
    cursor.execute(query)

    print('\nRaw encrypted values:')
    query = "select id, details from task"
    cursor.execute(query)
    for row in cursor.fetchall():
        print(row)

    print('\nDecrypting in query...')
    query = "select id, decrypt(details) from task"
    cursor.execute(query)
    for row in cursor.fetchall():
        print(row)

    print('\nDecrypting...')
    query = "update task set details = decrypt(details)"
    cursor.execute(query)
```

Functions are exposed using the `create_function()` method of the `Connection`. The parameters are the name of the function (as it should be used from within SQL), the number of arguments the function takes, and the Python function to expose.

```
$ python3 sqlite3_create_function.py

Original values:
(1, 'write about select')
(2, 'write about random')
(3, 'write about sqlite3')
(4, 'finish reviewing markup')
(5, 'revise chapter intros')
(6, 'subtitle')

Encrypting...
Encrypting 'write about select'
Encrypting 'write about random'
Encrypting 'write about sqlite3'
Encrypting 'finish reviewing markup'
Encrypting 'revise chapter intros'
Encrypting 'subtitle'

Raw encrypted values:
(1, 'jevgr nobhg fryrpg')
(2, 'jevgr nobhg enaqbz')
(3, 'jevgr nobhg fdyvgr3')
(4, 'svavfu erivrjvat znexhc')
(5, 'erivfr puncgre vagebf')
(6, 'fhogvgyr')

Decrypting in query...
Decrypting 'jevgr nobhg fryrpg'
Decrypting 'jevgr nobhg enaqbz'
Decrypting 'jevgr nobhg fdyvgr3'
Decrypting 'svavfu erivrjvat znexhc'
Decrypting 'erivfr puncgre vagebf'
```

```
Decrypting 'erivfr puncgre vagebf'
Decrypting 'fhogvgyr'
(1, 'write about select')
(2, 'write about random')
(3, 'write about sqlite3')
(4, 'finish reviewing markup')
(5, 'revise chapter intros')
(6, 'subtitle')

Decrypting...
Decrypting 'jevgr nobhg fryrpg'
Decrypting 'jevgr nobhg enaqbz'
Decrypting 'jevgr nobhg fdyvgr3'
Decrypting 'svavfu erivrjvat znexhc'
Decrypting 'erivfr puncgre vagebf'
Decrypting 'fhogvgyr'
```

## Querying with Regular Expressions

Sqlite supports several special user functions that are associated with SQL syntax. For example, a function regexp can be used in a query to check if a column's string value matches a regular expression using the following syntax.

```
SELECT * FROM table
WHERE column REGEXP '.*pattern.*'
```

This examples associates a function with `regexp()` to test values using Python's re module.

```
# sqlite3_regex.py

import re
import sqlite3

db_filename = 'todo.db'


def regexp(pattern, input):
    return bool(re.match(pattern, input))


with sqlite3.connect(db_filename) as conn:
    conn.row_factory = sqlite3.Row
    conn.create_function('regexp', 2, regexp)
    cursor = conn.cursor()

    pattern = '.*[wW]rite [aA]bout.*'

    cursor.execute(
        """
        select id, priority, details, status, deadline from task
        where details regexp :pattern
        order by deadline, priority
        """,
        {'pattern': pattern},
    )

    for row in cursor.fetchall():
        task_id, priority, details, status, deadline = row
        print('{:2d} [{:d}] {:<25} [{:<8}] ({})'.format(
            task_id, priority, details, status, deadline))
```

The output is all of the tasks where the details column matches the pattern.

```
$ python3 sqlite3_regex.py

 1 [9] write about select        [done    ] (2016-04-25)
 2 [9] write about random        [done    ] (2016-08-22)
 3 [9] write about sqlite3       [active  ] (2017-07-31)
```

## Custom Aggregation

## Custom Aggregation

An aggregation function collects many pieces of individual data and summarizes it in some way. Examples of built-in aggregation functions are `avg()` (average), `min()`, `max()`, and `count()`.

The API for aggregators used by `sqlite3` is defined in terms of a class with two methods. The `step()` method is called once for each data value as the query is processed. The `finalize()` method is called one time at the end of the query and should return the aggregate value. This example implements an aggregator for the arithmetic *mode*. It returns the value that appears most frequently in the input.

```
# sqlite3_create_aggregate.py

import sqlite3
import collections

db_filename = 'todo.db'


class Mode:

    def __init__(self):
        self.counter = collections.Counter()

    def step(self, value):
        print('step({!r})'.format(value))
        self.counter[value] += 1

    def finalize(self):
        result, count = self.counter.most_common(1)[0]
        print('finalize() -> {!r} ({} times)'.format(
            result, count))
        return result


with sqlite3.connect(db_filename) as conn:
    conn.create_aggregate('mode', 1, Mode)

    cursor = conn.cursor()
    cursor.execute("""
    select mode(deadline) from task where project = 'pymotw'
    """)
    row = cursor.fetchone()
    print('mode(deadline) is:', row[0])
```

The aggregator class is registered with the `create_aggregate()` method of the `Connection`. The parameters are the name of the function (as it should be used from within SQL), the number of arguments the `step()` method takes, and the class to use.

```
$ python3 sqlite3_create_aggregate.py

step('2016-04-25')
step('2016-08-22')
step('2017-07-31')
step('2016-11-30')
step('2016-08-20')
step('2016-11-01')
finalize() -> '2016-11-01' (1 times)
mode(deadline) is: 2016-11-01
```

## Threading and Connection Sharing

For historical reasons having to do with old versions of SQLite, `Connection` objects cannot be shared between threads. Each thread must create its own connection to the database.

```
# sqlite3_threading.py

import sqlite3
import sys
import threading
import time
```

```
db_filename = 'todo.db'
isolation_level = None  # autocommit mode


def reader(conn):
    print('Starting thread')
    try:
        cursor = conn.cursor()
        cursor.execute('select * from task')
        cursor.fetchall()
        print('results fetched')
    except Exception as err:
        print('ERROR:', err)


if __name__ == '__main__':
    with sqlite3.connect(db_filename,
                         isolation_level=isolation_level,
                         ) as conn:
        t = threading.Thread(name='Reader 1',
                             target=reader,
                             args=(conn,),
                             )
        t.start()
        t.join()
```

Attempts to share a connection between threads result in an exception.

```
$ python3 sqlite3_threading.py

Starting thread
ERROR: SQLite objects created in a thread can only be used in that
same thread.The object was created in thread id 140735234088960
and this is thread id 123145307557888
```

## Restricting Access to Data

Although SQLite does not have user access controls found in other, larger, relational databases, it does have a mechanism for limiting access to columns. Each connection can install an *authorizer function* to grant or deny access to columns at runtime based on any desired criteria. The authorizer function is invoked during the parsing of SQL statements, and is passed five arguments. The first is an action code indicating the type of operation being performed (reading, writing, deleting, etc.). The rest of the arguments depend on the action code. For `SQLITE_READ` operations, the arguments are the name of the table, the name of the column, the location in the SQL where the access is occurring (main query, trigger, etc.), and `None`.

```
# sqlite3_set_authorizer.py
```

```
import sqlite3

db_filename = 'todo.db'


def authorizer_func(action, table, column, sql_location, ignore):
    print('\nauthorizer_func({}, {}, {}, {}, {})'.format(
        action, table, column, sql_location, ignore))

    response = sqlite3.SQLITE_OK  # be permissive by default

    if action == sqlite3.SQLITE_SELECT:
        print('requesting permission to run a select statement')
        response = sqlite3.SQLITE_OK

    elif action == sqlite3.SQLITE_READ:
        print('requesting access to column {}.{} from {}'.format(
            table, column, sql_location))
        if column == 'details':
            print('  ignoring details column')
            response = sqlite3.SQLITE_IGNORE
        elif column == 'priority':
            print('  preventing access to priority column')
```

```
            response = sqlite3.SQLITE_DENY

    return response


with sqlite3.connect(db_filename) as conn:
    conn.row_factory = sqlite3.Row
    conn.set_authorizer(authorizer_func)

    print('Using SQLITE_IGNORE to mask a column value:')
    cursor = conn.cursor()
    cursor.execute("""
    select id, details from task where project = 'pymotw'
    """)
    for row in cursor.fetchall():
        print(row['id'], row['details'])

    print('\nUsing SQLITE_DENY to deny access to a column:')
    cursor.execute("""
    select id, priority from task where project = 'pymotw'
    """)
    for row in cursor.fetchall():
        print(row['id'], row['details'])
```

This example uses `SQLITE_IGNORE` to cause the strings from the `task.details` column to be replaced with null values in the query results. It also prevents all access to the `task.priority` column by returning `SQLITE_DENY`, which in turn causes SQLite to raise an exception.

```
$ python3 sqlite3_set_authorizer.py

Using SQLITE_IGNORE to mask a column value:

authorizer_func(21, None, None, None, None)
requesting permission to run a select statement

authorizer_func(20, task, id, main, None)
requesting access to column task.id from main

authorizer_func(20, task, details, main, None)
requesting access to column task.details from main
  ignoring details column

authorizer_func(20, task, project, main, None)
requesting access to column task.project from main
1 None
2 None
3 None
4 None
5 None
6 None

Using SQLITE_DENY to deny access to a column:

authorizer_func(21, None, None, None, None)
requesting permission to run a select statement

authorizer_func(20, task, id, main, None)
requesting access to column task.id from main

authorizer_func(20, task, priority, main, None)
requesting access to column task.priority from main
  preventing access to priority column
Traceback (most recent call last):
  File "sqlite3_set_authorizer.py", line 53, in <module>
    """)
sqlite3.DatabaseError: access to task.priority is prohibited
```

The possible action codes are available as constants in `sqlite3`, with names prefixed `SQLITE_`. Each type of SQL statement can be flagged, and access to individual columns can be controlled as well.

## See also

## See also

- Standard library documentation for sqlite3
- **PEP 249** – DB API 2.0 Specification (A standard interface for modules that provide access to relational databases.)
- SQLite – The official site of the SQLite library.
- `shelve` – Key-value store for saving arbitrary Python objects.
- SQLAlchemy – A popular object-relational mapper that supports SQLite among many other relational databases.

dbm — Unix Key-Value Databases

xml.etree.ElementTree — XML Manipulation API

**Quick Links**

Creating a Database
Retrieving Data
Query Metadata
Row Objects
Using Variables with Queries
Positional Parameters
Named Parameters
Bulk Loading
Defining New Column Types
Determining Types for Columns
Transactions
Preserving Changes
Discarding Changes
Isolation Levels
Deferred
Immediate
Exclusive
Autocommit
In-Memory Databases
Exporting the Contents of a Database
Using Python Functions in SQL
Querying with Regular Expressions
Custom Aggregation
Threading and Connection Sharing
Restricting Access to Data

*This page was last updated 2017-01-06.*

**Navigation**

dbm — Unix Key-Value Databases
xml.etree.ElementTree — XML Manipulation API



*The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.*

*Looking for examples for Python 2?*

**Other Writing**

- Blog
- The Python Standard Library By Example