Python and SQLite Gotchas and Gimmes

PyCon Presentation Apr 2016

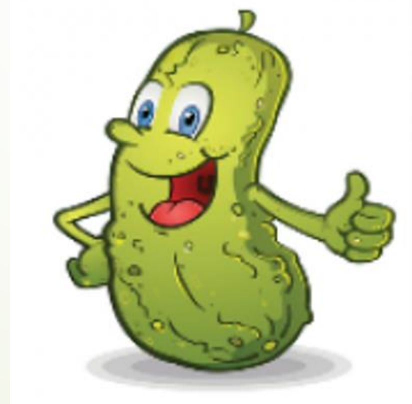
Dave Sawyer
dsawyer@box.com

What is SQLite?

- In-process SQL database engine
- Free for any use
- Compact (300K lib, 4K stack, 100K heap)
- Fast (10,000+ statements / second)
- Reliable (100% branch coverage, 787x test code)

Why use SQLite?

We have pickle. Isn't that good enough?



Why use SQLite?

- Handle large datasets
- Sort/Search efficiently
- Transactions with rollback
- Safe format
- 32/64 big/little endian agnostic
- Concurrency

Simple SQLite

```
# Copied directly from Python documentation on sqlite3
import sqlite3
connection = sqlite3.connect('example.db')
c = connection.cursor()

# Create table
c.execute("CREATE TABLE stocks (symbol text, quantity real, price real)")

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('RHAT', 100, 35.14)")

# Save (commit) the changes
connection.commit()

# We can close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost
connection.close()
```

Simple?

```
# Copied directly from Python documentation on sqlite3
import sqlite3
connection = sqlite3.connect('example.db')
c = connection.cursor()

# Create table
c.execute("CREATE TABLE stocks (symbol text, quantity real, price real)")

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('RHAT', 100, 35.14)")

# Save (commit) the changes
connection.commit()

# We can close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost
connection.close()
```

Use Context Managers

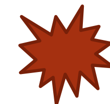
```python
import sqlite3
connection = sqlite3.connect('example.db')

# Create table
## Use a transaction to commit or rollback a set of changes
with connection:
    connection.execute("CREATE TABLE stocks (symbol text, quantity real, price real)")

# Insert a row of data
with connection:
    connection.execute("INSERT INTO stocks VALUES ('RHAT', 100, 35.14)")

# We can close the connection if we are done with it.
connection.close()
```

Context manager



Cursor “convenience”

Getting Real

```
def sql_value(value):
    """Convert a python value to something that can be stored in SQLite"""
    # Note: Not needed if we update code to use DBAPI bindings.
    if isinstance(value, basestring):
        return "'" + value + "'"
    else:
        return unicode(value)


class Stock(object):
    """Represents a stock holding (symbol, quantity, and price"""
    def __init__(self, symbol='', quantity=0, price=0.0):
        self.symbol = symbol
        self.quantity = quantity
        self.price = price

    @classmethod
    def from_row(cls, row):
        return Stock(*row)
```

Getting Real

```python
class StockDB(object):
    def __init__(self):
        self._connection = sqlite3.connect('example.db')
```

Getting Real

```python
class StockDB(object):
    def __init__(self):
        self._connection = sqlite3.connect('example.db')
```

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('RHAT', 100, 35.14)")

```
def insert(self, stock):
        keys = stock.__dict__.iterkeys()
        values = (sql_value(x) for x in stock.__dict__.itervalues())
        with closing(self._connection.cursor()) as cursor:
            cursor.execute("INSERT INTO stocks({}) VALUES ({})".format(
                                ", ".join(keys), ", ".join(values)))
```

Unsafe!

HTTP://XKCD.COM/327/

Example Code (take 3)

```python
import sqlite3
connection = sqlite3.connect('example.db')

# Create table
## Use a transaction to commit or rollback a set of changes
with connection:
    connection.execute("CREATE TABLE stocks (symbol text, quantity real, price real)")

# Insert a row of data
with connection:
    connection.execute("INSERT INTO stocks VALUES (?,?,?)", ('RHAT', 100, 35.14))

# We can close the connection if we are done with it.
connection.close()
```

Getting Real

```
class StockDB(object):
    def __init__(self):
        self._connection = sqlite3.connect('example.db')

    def create_table(self):
        with closing(self._connection.cursor()) as cursor:
            cursor.execute("CREATE TABLE stocks (symbol text, quantity real, price real)")
```

```
    def insert(self, stock):
        places = ','.join(['?'] * len(stock.__dict__))
        keys = ','.join(stock.__dict__.iterkeys())
        values = tuple(stock.__dict__.itervalues())
        with closing(self._connection.cursor()) as cursor:
            cursor.execute("INSERT INTO stocks({}) VALUES ({})".format(
                                keys, places), values)
```

Getting Real

```
class StockDB(object):
    def __init__(self):
        self._connection = sqlite3.connect('example.db')

    def create_table(self):
        with closing(self._connection.cursor()) as cursor:
            cursor.execute("CREATE TABLE stocks (symbol text, quantity real, price real)")

    def insert(self, stock):
        places = ','.join(['?'] * len(stock.__dict__))
        keys = ','.join(stock.__dict__.iterkeys())
        values = tuple(stock.__dict__.itervalues())
        with closing(self._connection.cursor()) as cursor:
            cursor.execute("INSERT INTO stocks({}) VALUES ({})".format(keys, places), values)
```

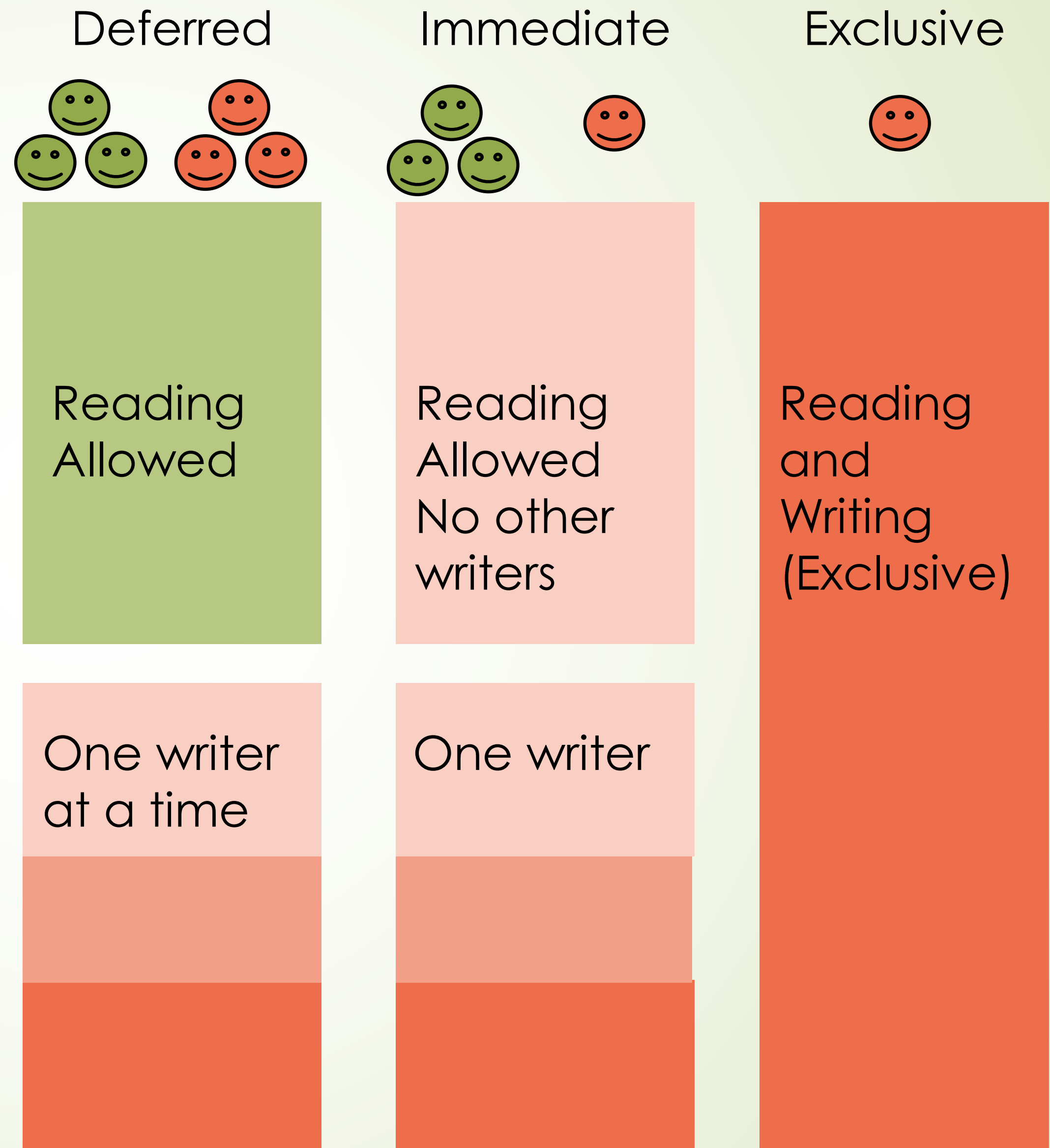
```
    def lookup(self, symbol):
        with closing(self._connection.cursor()) as cursor:
            cursor.execute("SELECT * FROM stocks WHERE symbol= ?", (symbol,))
            row = cursor.fetchone()
            return Stock.from_row(row) if row else None
```

Getting Real

```
class StockDB(object):
    def __init__(self):
        self._connection = sqlite3.connect('example.db')
```

```
    def transaction(self):
        return self._connection
```

SQLite Isolation Levels



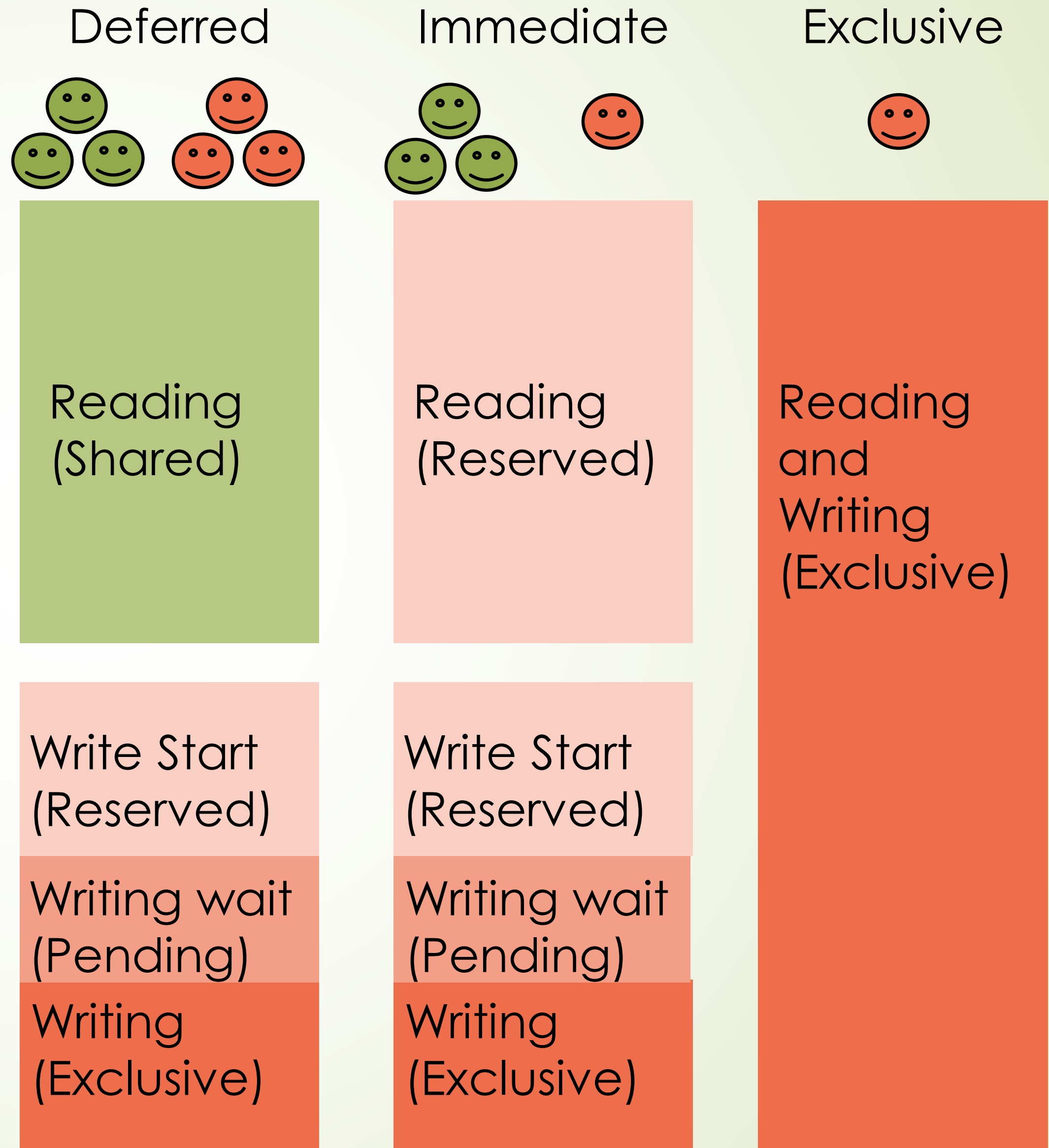
SQLite Isolation Levels - Locks

Shared Lock
reading

Reserved Lock
Only 1 of these.
Shared still ok.
Setup journal.

Pending Lock
No more shared locks given out.
Draining reads.

Exclusive Lock
No locks of any kind given out.



Getting Concurrent

```
class StockDB(object):
    def __init__(self):
        ## 1. ADD check_same_thread
        ## This allows us to use multiple threads on the same connection.
        ## Requires SQLite 3.3.1 (Jan 2006) or later
        ## 2. Change the isolation level to deferred so we can control transactions
        self._connection = sqlite3.connect('example.db', check_same_thread=False,
isolation_level='DEFERRED')
        ## 3. Use WAL mode. Requires SQLite 3.7.0 (Jul 2010)
        self._connection.execute('PRAGMA journal_mode = WAL')
```

`sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements])`

Opens a connection to the SQLite database file *database*. You can use `":memory:"` to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The *timeout* parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the *isolation_level* parameter, please see the `Connection.isolation_level` property of `Connection` objects.

SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The *detect_types* parameter and the using custom **converters** registered with the module-level `register_converter()` function allow you to easily do that.

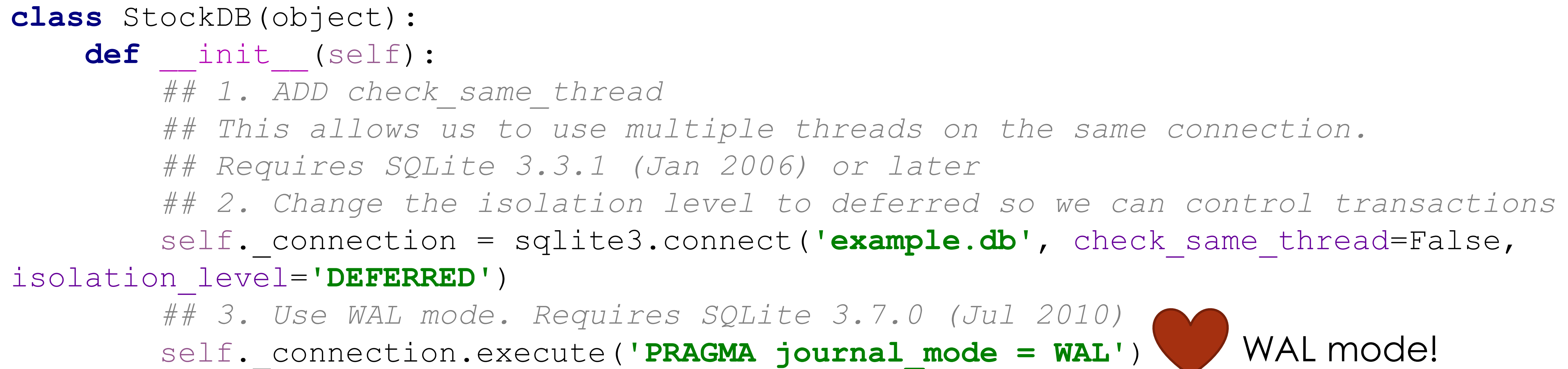
detect_types defaults to 0 (i. e. off, no type detection), you can set it to any combination of `PARSE_DECLTYPES` and `PARSE_COLNAMES` to turn type detection on.

By default, the `sqlite3` module uses its `Connection` class for the connect call. You can, however, subclass the `Connection` class and make `connect()` use your class instead by providing your class for the *factory* parameter.

Consult the section *SQLite and Python types* of this manual for details.

The `sqlite3` module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the *cached_statements* parameter. The currently implemented default is to cache 100 statements.

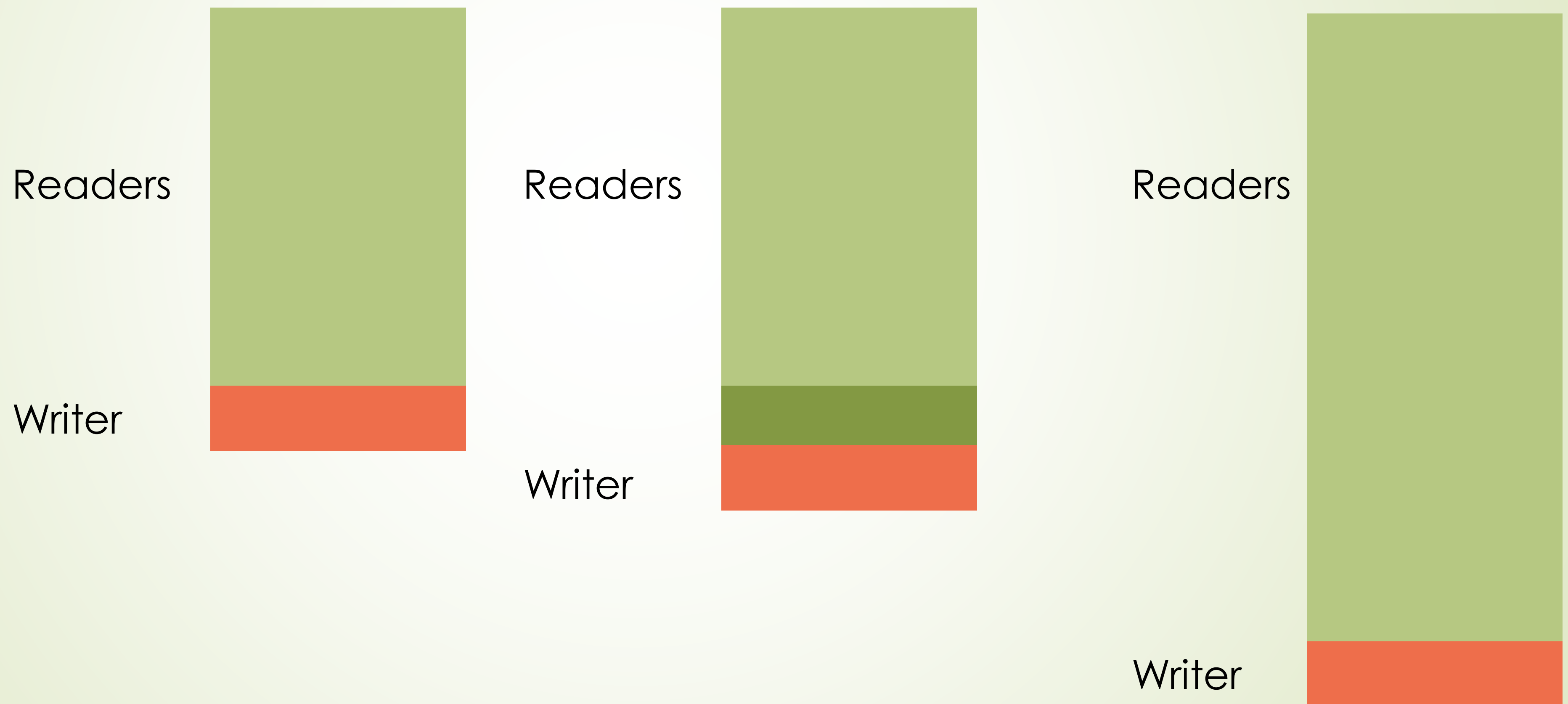
Getting Concurrent



```
class StockDB(object):
    def __init__(self):
        ## 1. ADD check_same_thread
        ## This allows us to use multiple threads on the same connection.
        ## Requires SQLite 3.3.1 (Jan 2006) or later
        ## 2. Change the isolation level to deferred so we can control transactions
        self._connection = sqlite3.connect('example.db', check_same_thread=False,
isolation_level='DEFERRED')
        ## 3. Use WAL mode. Requires SQLite 3.7.0 (Jul 2010)
        self._connection.execute('PRAGMA journal_mode = WAL')
```

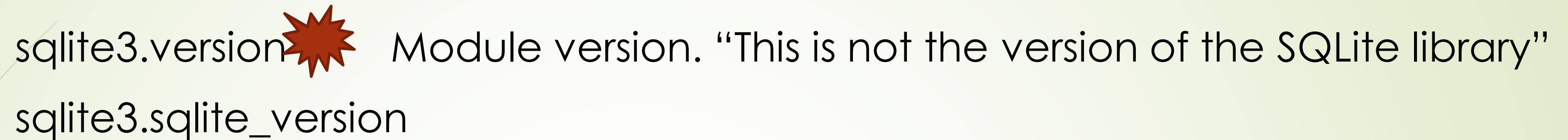
What is WAL mode?

Write-Ahead Logging



This lets us read at the same time we write

How Do I Get This?



sqlite3.version Module version. “This is not the version of the SQLite library”

sqlite3.sqlite_version

Platform	Version
Windows, Python 2.7.x	3.6.21
Windows, Python 3.4.x	3.8.3.1
Windows, Python 3.5.1	3.8.11
OSX 10.11	3.8.10.2
OSX 10.10	3.8.5
OSX 10.9	3.7.13
OSX 10.8	3.6.18
Linux	Check manually

To upgrade, just drop in new version of SQLite (sqlite.dll or /usr/bin/sqlite3)

Next Steps?

Only one set of changes per connection

- Create a connection per thread

```
@property
def connection(self):
    db_connection = thread_connections.get(get_ident(), None)
    if db_connection is None:
        db_connection = self._sqlite_connect()
        thread_connections[get_ident()] = db_connection
```

- Use a lock when writing

```
@contextmanager
def transaction(self):
    with self._lock:
        try:
            yield
            self._connection.commit()
        except:
            self._connection.rollback()
            raise
```

Both work!

- Complete code available on github.com/kingsawyer
- References:
  https://docs.python.org/2/library/sqlite3.html
  https://www.sqlite.org/lockingv3.html