swee'pea Documentation

Release 0.1.0

charles leifer

Jul 17, 2018

Contents

1 Installation **3**
1.1 Installing with git 3
1.2 Running tests 3

2 Swee'pea's API **5**
2.1 Database 7
2.2 SQL Builder 13

3 Indices and tables **17**

Fast, lightweight Python database toolkit for SQLite, built with Cython.

Like it's cousin peewee, `swee'pea` is comprised of a database connection abstraction and query-building / execution APIs. This project is a pet project of mine, so tailor expectations accordingly.

Features:

- Implemented in Cython for performance and to expose advanced features of the SQLite database library.
- Composable and consistent APIs for building queries using Python.
- Layered APIs allow you to work as close to the database as you want.
- No magic.
- No bullshit.

Issue tracker and code are hosted on GitHub: https://github.com/coleifer/sweepea.

Documentation hosted on RT**F**D: https://sweepea.readthedocs.io/

Contents:

CHAPTER 1

Installation

Most users will want to simply install the latest version, hosted on PyPI:

```
pip install sweepea
```

You'll need to ensure that Cython is installed in order to compile the shared library. Furthermore, your Python will need to be compiled with support for `sqlite3` **or** you'll need to install `pysqlite`.

1.1 Installing with git

The project is hosted at https://github.com/coleifer/sweepea and can be installed using git:

```
git clone https://github.com/coleifer/sweepea.git
cd sweepea
python setup.py build
python setup.py install
```

Note: On some systems you may need to use `sudo python setup.py install` to install peewee system-wide.

1.2 Running tests

You can test your installation by running the test suite.

```
python setup.py test

# Or run the test module directly:
python tests.py
```

CHAPTER 2

Swee'pea's API

class TableFunction

Implements a table-valued function (eponymous-only virtual table) in SQLite. In English, a table-valued function is a user-defined function that can return 0 or more rows of data. Normal user-defined functions must return a scalar value, and aggregate functions can accept multiple inputs but are still restricted to a single scalar output. These restrictions are lifted from table-valued functions. They are called table-valued because their output can be thought of as a table.

Subclasses must define the `columns` (return values) and `params` (input values) to their function. These are declared as class attributes.

Subclasses must also implement two methods:

- `initialize(**query)`
- `iterate(idx)`

```
# Example table-valued function that returns a range of integers.
class Series(TableFunction):
    columns = ['value']
    params = ['start', 'stop', 'step']
    name = 'series'

    def initialize(self, start=0, stop=None, step=1):
        self.start = self.current = start
        self.stop = stop or float('inf')
        self.step = step

    def iterate(self, idx):
        if self.current > self.stop:
            raise StopIteration

        return_val = self.current
        self.current += self.step
        return (return_val,)
```

(continues on next page)

(continued from previous page)

```
# Must register with a connection in order to use.
conn = sqlite3.connect(':memory:')
Series.register(conn)

# Now we can call it.
for num, in conn.execute('select * from series(0, 10, 2)'):
    print num

# Prints 0, 2, 4, 6, 8, 10.
```

columns
A list or tuple describing the rows returned by this function.

params
A list or tuple describing the parameters this function accepts.

name
The name of the table-valued function. If not provided, name will be inferred from the class name.

initialize(***query*)
This method is called once for each set of values the table-valued function is called with.

Parameters **query** – The parameters the function was called with.

Returns return value is discarded.

iterate(*idx*)
This method is called repeatedly until it raises `StopIteration`. The return value should be a row-tuple matching the format described by the `columns` attribute.

Parameters **idx** (*int*) – The row index being requested.

Raises StopIteration

Returns A row tuple

register(*connection*)
Register the table-valued function with a SQLite database connection. You must register a table-valued function in order to use it.

Parameters **connection** – a `sqlite3.Connection` instance.

class CursorWrapper(*cursor*)
Wraps a SQLite3 cursor, providing additional functionality. This object should not be instantiated directly, but instead is returned when executing `SELECT` queries.

When iterated over, the cursor wrapper will yield result rows as tuples.

iterator()
Provides an iterator over the result-set that does not cache the result rows. Use this for iterating over large result sets, or result sets that only need to be iterated over once.

Example:

```
# Query will return a large number of rows.
query = PageView.select(PageView.url, PageView.timestamp)
for row in query.execute(db).iterator():
    report.write((row.url, row.timestamp))
```

first()
Return the first row or `None` if no rows were returned.

get()
Return the first row or raise a `DoesNotExist` exception if no rows were returned.

Raises DoesNotExist

scalar()
Returns the first column of the first row, or raise a `DoesNotExist` if no rows were returned. Useful for retrieving the value of a query that performed an aggregation, like a `COUNT()` or `SUM()`.

class DictCursorWrapper
A subclass of *CursorWrapper* that yields result rows as dictionaries.

class NamedTupleCursorWrapper
A subclass of *CursorWrapper* that yields result rows as named tuples.

class ObjectCursorWrapper(*cursor*, *constructor*)
A subclass of *CursorWrapper* that accepts a constructor and for each result tuple, will call the constructor with the row and yield the return value.

Parameters **constructor** – A callable which accepts a row of data and returns an arbitrary object.

2.1 Database

class Database(*database*[, *pragmas=None*[, *journal_mode=None*[, *rank_functions=False*[, *regex_function=True*[, *hash_functions=False*[, ***kwargs*]]]]]])
Wrapper for managing SQLite database connections. Handles connections in a thread-safe manner and provides Pythonic APIs for managing transactions, executing queries, and introspecting database internals.

Parameters

- **database** – The filename of the SQLite database, or the string `':memory:'` for an in-memory database. To defer the initialization of the database, you can also specify `None`.
- **pragmas** – A list of 2-tuples describing the pragma key and value to be applied when a connection is opened.
- **journal_mode** – Journaling mode to use with SQLite database.
- **rank_functions** (*bool*) – Whether to register user-defined functions for scoring search results. For use with full-text-search extension.
- **regex_function** (*bool*) – Whether to register a user-defined function to provide support for the `REGEXP` operator.
- **hash_functions** (*bool*) – Whether to register cryptographic hash functions.
- **kwargs** – Arbitrary keyword arguments passed to the `sqlite3` connection constructor.

init(*database*, ***connect_kwargs*)
This method is used to initialize a deferred database. A database is said to be deferred when it is instantiated with the database file as `None`. Reasons you might do this are to declare the database in one place, and actually assign it to a given file elsewhere in the code (e.g. for running tests).

Parameters

- **database** – The filename of the SQLite database, or the string `':memory:'` for an in-memory database.
- **connect_kwargs** – Arbitrary keyword arguments passed to the `sqlite3` connection constructor.

connect([*reuse_if_open=False*])

Open a connection to the SQLite database. If a connection already exists for the current thread, an `OperationalError` will be raised. Alternatively, you can specify `reuse_if_open` to suppress the error in the event a connection is already open.

Parameters **reuse_if_open** (*bool*) – If a connection already exists, re-use it rather than raising an exception.

Raises **OperationalError** –

Rtype bool

Returns Boolean value indicating whether a connection was opened. Will always be `True` unless `reuse_if_open` was specified and a connection already existed.

close()

Close the current thread's connection. If no connection is currently open, no exception will be raised.

Rtype bool

Returns Boolean indicating whether a connection was closed.

aggregate([*name=None*])

Decorator for declaring and registering a user-defined aggregate function.

Example:

```
@db.aggregate('avg')
class Average(object):
    def __init__(self):
        self.vals = []

    def step(self, value):
        self.vals.append(value)

    def finalize(self):
        return sum(self.vals) / len(self.vals)
```

func([*name=None*[, *n=-1*[, *deterministic=True*]]])

Decorator for declaring and registering a user-defined function. User-defined functions accept up to `n` parameters and return a scalar value. If `n` is not fixed, you may specify `-1`.

Parameters

- **name** (*str*) – Name of the function.
- **n** (*int*) – Number of parameters function accepts, or `-1`.
- **deterministic** (*bool*) – Function is deterministic.

Example:

```
@db.func('md5')
def md5(s):
    return hashlib.md5(s).hexdigest()
```

table_function([*name=None*])

Decorator for declaring and registering a table-valued function with the database. Table-valued functions are described in the section on *TableFunction*, but briefly, a table-valued function accepts any number of parameters, and instead of returning a scalar value, returns any number of rows of tabular data.

Example:

```
@db.table_function('series')
class Series(TableFunction):
    columns = ['value']
    params = ['start', 'stop']

    def initialize(self, start=0, stop=None):
        self.start, self.stop = start, (stop or float('Inf'))
        self.current = self.start

    def iterate(self, idx):
        if self.current > self.stop:
            raise StopIteration
        ret = self.current
        self.current += 1
        return (ret,)
```

on_commit (*func*)

Decorator for declaring and registering a post-commit hook. The handler's return value is ignored, but if a `ValueError` is raised, then the transaction will be rolled-back.

The decorated function should not accept any parameters.

Example:

```
@db.on_commit
def commit_handler():
    if datetime.date.today().weekday() == 6:
        raise ValueError('no commits on sunday!')
```

on_rollback (*func*)

Decorator for registering a rollback handler. The return value is ignored.

The decorated function should not accept any parameters.

Example:

```
@db.on_rollback
def rollback_handler():
    logger.info('rollback was issued.')
```

on_update (*func*)

Decorator for registering an update hook. The decorated function is executed for each row that is inserted, updated or deleted. The return value is ignored.

User-defined callback must accept the following parameters:

- query type (INSERT, UPDATE or DELETE)
- database name (typically 'main')
- table name
- rowid of affected row

Example:

```
@db.on_update
def change_logger(query_type, db, table, rowid):
    logger.debug('%s query on %s.%s row %s', query_type, db,
                 table, rowid)
```

is_closed()
Return a boolean indicating whether the database is closed.

connection()
Get the currently open connection. If the database is closed, then a new connection will be opened and returned.

execute_sql(*sql*[, *params=None*[, *commit=True*]])
Execute the given SQL query and returns the cursor. If no connection is currently open, one will be opened automatically.

Parameters

- **sql** – SQL query
- **params** – A list or tuple of parameters for the query.
- **commit** (*bool*) – Whether a `commit` should be invoked after the query is executed.

Returns A `sqlite3.Cursor` instance.

execute(*query*)
Execute the SQL query represented by the `Query` object. The query will be parsed into a parameterized SQL query automatically.

Parameters **query** (*Query*) – The `Query` instance to execute.

Returns A `sqlite3.Cursor` instance.

pragma(*key*[, *value=SENTINEL*])
Issue a PRAGMA query on the current connection. To query the status of a specific PRAGMA, typically only the `key` will be specified.

For more information, see the SQLite PRAGMA docs.

Note: Many `PRAGMA` settings are exposed as properties on the *Database* object.

---

add_pragma(*key*, *value*)
Apply the specified pragma query each time a new connection is opened. If a connection is currently open, the pragma will be executed.

remove_pragma(*key*)
Remove the pragma operation specified by the given key from the list of pragma queries executed on each new connection.

begin([*lock_type=None*])
Start a transaction using the specified lock type. If the lock type is unspecified, then a bare `BEGIN` statement is issued.

Because swee'pea runs `sqlite3` in autocommit mode, it is necessary to explicitly begin transactions using this method.

For an alternative API, see the *atomic()* helper.

commit()
Call `commit()` on the currently-open `sqlite3.Connection` object.

rollback()
Call `rollback()` on the currently-open `sqlite3.Connection` object.

__getitem__(*name*)
Factory method for creating *BoundTable* instances.

Example:

```
UserTbl = db['users']
query = UserTbl.select(UserTbl.c.username)
for username, in query.execute():
    print username
```

__enter__()
Use the database as a context-manager. When the context manager is entered, a connection is opened (if one is not already open) and a transaction begins. When the context manager exits, the transaction is either committed or rolled-back (depending on whether the context manager exits with an exception). Finally, the connection is closed.

Example:

```
with database:
    database.execute_sql('CREATE TABLE foo (data TEXT)')
    FooTbl = database['foo']
    for i in range(10):
        FooTbl.insert({FooTbl.c.data: str(i)}).execute()
```

last_insert_rowid()
Return the `rowid` of the most-recently-inserted row on the currently active connection.

changes()
Return the number of rows changed by the most recent query.

autocommit
A property which indicates whether the connection is in autocommit mode or not.

set_busy_handler([*timeout=5000*])
Replace the default SQLite busy handler with one that introduces some *jitter* into the amount of time delayed between checks. This addresses an issue that frequently occurs when multiple threads are attempting to modify data at nearly the same time.

Parameters **timeout** – Max number of milliseconds to attempt to execute query.

atomic()
Context manager or decorator that executes the wrapped statements in either a transaction or a savepoint. The outer-most call to `atomic` will use a transaction, and any subsequent nested calls will use savepoints.

Note: For most use-cases, it makes the most sense to always use `atomic` when you wish to execute queries in a transaction. The benefit of using `atomic` is that you do not need to manually keep track of the transaction stack depth, as this will be managed for you.

---

transaction()
Execute statements in a transaction using either a context manager or decorator. If an error is raised inside the wrapped block, the transaction will be rolled back, otherwise statements are committed when exiting. Transactions can also be explicitly rolled back or committed within the transaction block by calling `rollback()` or `commit()`. If you manually commit or roll back, a new transaction will be started automatically.

Nested blocks can be wrapped with `transaction` - the database will keep a stack and only commit when it reaches the end of the outermost function / block.

savepoint()
Execute statements in a savepoint using either a context manager or decorator. If an error is raised inside the wrapped block, the savepoint will be rolled back, otherwise statements are committed when exiting. Like

transaction(), a savepoint can also be explicitly rolled-back or committed by calling `rollback()` or `commit()`. If you manually commit or roll back, a new savepoint **will not** be created.

Savepoints can be thought of as nested transactions.

Parameters **sid** (*str*) – An optional string identifier for the savepoint.

get_tables()
Return a sorted list of the tables in the database.

get_indexes(*table*)
Returns a list of index metadata for the given table. The index metadata is returned as a 4-tuple consisting of:

- Index name.
- SQL used to create the index.
- Names of columns being indexed.
- Whether the index is unique.

get_columns(*table*)
Returns a list of column metadata for the given table. Column metadata is returned as a 4-tuple consisting of:

- Column name.
- Data-type column was declared with.
- Whether the column can be NULL.
- Whether the column is the primary key.

get_primary_keys(*table*)
Returns a list of column(s) that comprise the table's primary key.

get_foreign_keys(*table*)
Returns a list of foreign key metadata for the given table. Foreign key metadata is returned as a 3-tuple consisting of:

- Source column name, i.e. the column on the given table.
- Destination table.
- Destination column.

backup(*dest_db*)
Backup the current database to the given destination *Database* instance.

Parameters **dest_db** (*Database*) – database to hold backup.

backup_to_file(*filename*)
Backup the current database to the given filename.

cache_size
Property that exposes `PRAGMA cache_size`.

foreign_keys
Property that exposes `PRAGMA foreign_keys`.

journal_mode
Property that exposes `PRAGMA journal_mode`.

journal_size_limit
Property that exposes `PRAGMA journal_size_limit`.

mmap_size
Property that exposes `PRAGMA mmap_size`.

page_size
Property that exposes `PRAGMA page_size`

read_uncommited
Property that exposes `PRAGMA read_uncommited`

synchronous
Property that exposes `PRAGMA synchronous`

wal_autocheckpoint
Property that exposes `PRAGMA wal_autocheckpoint`

2.2 SQL Builder

class Table (*name*[, *columns=None*[, *schema=None*[, *alias=None*]]])
Represents a table in a SQL query. Tables can be initialized with a static list of columns, or columns can be referenced dynamically using the `c` attribute.

Example:

```
# Example using a static list of columns:
UserTbl = Table('users', ('id', 'username'))
query = (UserTbl
         .select()  # "id" and "username" automatically selected.
         .order_by(UserTbl.username))

# Using dynamic columns:
TweetTbl = Table('tweets')
query = (TweetTbl
         .select(TweetTbl.c.content, TweetTbl.c.timestamp)
         .join(UserTbl, on=(TweetTbl.c.user_id == UserTbl.id))
         .where(UserTbl.username == 'charlie')
         .order_by(TweetTbl.c.timestamp.desc()))
```

bind (*database*)
Create a table reference that is bound to the given database. Returns a *BoundTable* instance.

select (**selection*)
Create a `Select` query from the given table. If the `selection` is not provided, and the table defines a static list of columns, then the selection will default to all defined columns.

Parameters **selection** – values to select.

Returns a `Select` query instance.

insert ([*data=None*[, *columns=None*[, *on_conflict=None*[, ***kwargs*]]]])
Create a `Insert` query into the given table. Data to be inserted can be provided in a number of different formats:

- a dictionary of table columns to values
- keyword arguments of column names to values
- a list/tuple/iterable of dictionaries
- a `Select` query instance.

Note: When providing a `Select` query, it is necessary to also provide a list of columns.

It is also advisable to provide a list of columns when supplying a list or iterable of rows to insert.

Simple insert example:

```
User = Table('users', ('id', 'username', 'is_admin'))

# Simple inserts.
query = User.insert({User.username: 'huey', User.is_admin: False})

# Equivalent to above.
query = User.insert(username='huey', is_admin=False)
```

Inserting multiple rows:

```
# Inserting multiple rows of data.
data = (
    {User.username: 'huey', User.is_admin: False},
    {User.username: 'mickey', User.is_admin: True})
query = User.insert(data)

# Equivalent to above
query = User.insert(data, columns=(User.username, User.is_admin))
```

Inserting using a SELECT query:

```
Person = Table('person', ('id', 'name'))

query = User.insert(
    Person.select(Person.name, False),
    columns=(User.username, User.is_admin))

# Results in:
# INSERT INTO "users" ("username", "is_admin")
# SELECT "person"."name", false FROM "person";
```

update([*data=None*[, *on_conflict=None*[, ***kwargs*]]])

Create a `Update` query for the given table. Update can be provided as a dictionary keyed by column, or using keyword arguments.

Examples:

```
User = Table('users', ('id', 'username', 'is_admin'))

query = (User
         .update({User.is_admin: False})
         .where(User.username == 'huey'))

# Equivalent to above:
query = User.update(is_admin=False).where(User.username == 'huey')
```

Example of an atomic update:

```
PageView = Table('pageviews', ('url', 'view_count'))

query = (PageView
```

(continues on next page)

(continued from previous page)

```
        .update({PageView.view_count: PageView.view_count + 1})
        .where(PageView.url == some_url))
```

delete()
Create a `Delete` query for the given table.

Example:

```
query = User.delete().where(User.c.account_expired == True)
```

filter(***kwargs*)
Perform a `Select` query, filtering the result set using keyword arguments to represent filter expressions. All expressions are combined using `AND`.

This method is provided as a convenience API.

Example:

```
Person = Table('person', ('id', 'name', 'dob'))

today = datetime.date.today()
eighteen_years_ago = today - datetime.timedelta(years=18)
adults = Person.filter(dob__gte=eighteen_years_ago)
```

rank()
Convenience method for representing an expression which calculates the rank of search results.

Example:

```
NoteIdx = Table('note_idx', ('docid', 'content'))
rank = NoteIdx.rank()
query = (NoteIdx
         .select(NoteIdx.docid, NoteIdx.content, rank.alias('score'))
         .where(NoteIdx.match('search query'))
         .order_by(rank)
         .namedtuples())

for search_result in query.execute(database):
    print search_result.score, search_result.content
```

bm25()
Convenience method for representing an expression which calculates the rank of search results using the BM25 algorithm. Usage is identical to *rank()*.

match(*search_term*)
Convenience method for generating an expression that corresponds to a search on a full-text search virtual table. For an example of usage, see *rank()*.

class BoundTable(*database*, *name*[, *columns=None*[, *schema=None*[, *alias=None*]]])
Identical to *Table* with the exception that any queries on the table will automatically be bound to the provided database.

With an ordinary *Table* object, you would write the following:

```
User = Table('users', ('id', 'username'))
query = User.select().namedtuples()
for user in query.execute(database):
    print user.username
```

With a bound table, you can instead write:

```
BoundUser = User.bind(database)
query = User.select().namedtuples()
for user in query.execute():
    print user.username

# Or, even simpler, since bound select queries implement __iter__:
for user in query:
    print user.username
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__enter__() (Database method), 11
__getitem__() (Database method), 10

A

add_pragma() (Database method), 10
aggregate() (Database method), 8
atomic() (Database method), 11
autocommit (Database attribute), 11

B

backup() (Database method), 12
backup_to_file() (Database method), 12
begin() (Database method), 10
bind() (Table method), 13
bm25() (Table method), 15
BoundTable (built-in class), 15

C

cache_size (Database attribute), 12
changes() (Database method), 11
close() (Database method), 8
columns (TableFunction attribute), 6
commit() (Database method), 10
connect() (Database method), 7
connection() (Database method), 10
CursorWrapper (built-in class), 6

D

Database (built-in class), 7
delete() (Table method), 15
DictCursorWrapper (built-in class), 7

E

execute() (Database method), 10
execute_sql() (Database method), 10

F

filter() (Table method), 15
first() (CursorWrapper method), 6
foreign_keys (Database attribute), 12
func() (Database method), 8

G

get() (CursorWrapper method), 6
get_columns() (Database method), 12
get_foreign_keys() (Database method), 12
get_indexes() (Database method), 12
get_primary_keys() (Database method), 12
get_tables() (Database method), 12

I

init() (Database method), 7
initialize() (TableFunction method), 6
insert() (Table method), 13
is_closed() (Database method), 9
iterate() (TableFunction method), 6
iterator() (CursorWrapper method), 6

J

journal_mode (Database attribute), 12
journal_size_limit (Database attribute), 12

L

last_insert_rowid() (Database method), 11

M

match() (Table method), 15
mmap_size (Database attribute), 12

N

name (TableFunction attribute), 6
NamedTupleCursorWrapper (built-in class), 7

O

ObjectCursorWrapper (built-in class), 7
on_commit() (Database method), 9
on_rollback() (Database method), 9

on_update() (Database method), 9

P

page_size (Database attribute), 13
params (TableFunction attribute), 6
pragma() (Database method), 10

R

rank() (Table method), 15
read_uncommited (Database attribute), 13
register() (TableFunction method), 6
remove_pragma() (Database method), 10
rollback() (Database method), 10

S

savepoint() (Database method), 11
scalar() (CursorWrapper method), 7
select() (Table method), 13
set_busy_handler() (Database method), 11
synchronous (Database attribute), 13

T

Table (built-in class), 13
table_function() (Database method), 8
TableFunction (built-in class), 5
transaction() (Database method), 11

U

update() (Table method), 14

W

wal_autocheckpoint (Database attribute), 13