peewee Documentation

Release 3.13.1

charles leifer

Mar 09, 2020

Contents

Peewee is a simple and small ORM. It has few (but expressive) concepts, making it easy to learn and intuitive to use.

- a small, expressive ORM
- python 2.7+ and 3.4+ (developed with 3.6)
- supports sqlite, mysql, postgresql and cockroachdb
- *tons of extensions*

Peewee's source code hosted on GitHub.

New to peewee? These may help:

- *Quickstart*
- *Example twitter app*
- *Using peewee interactively*
- *Models and fields*
- *Querying*
- *Relationships and joins*

CHAPTER 1

Contents:

1.1 Installing and Testing

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

```
pip install peewee
```

Peewee comes with a couple C extensions that will be built if Cython is available.

- Sqlite extensions, which includes Cython implementations of the SQLite date manipulation functions, the REGEXP operator, and full-text search result ranking algorithms.

1.1.1 Installing with git

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

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

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

If you would like to build the SQLite extension in a git checkout, you can run:

```
# Build the C extension and place shared libraries alongside other modules.
python setup.py build_ext -i
```

1.1.2 Running tests

You can test your installation by running the test suite.

```
python runtests.py
```

You can test specific features or specific database drivers using the `runtests.py` script. To view the available test runner options, use:

```
python runtests.py --help
```

Note: To run tests against Postgres or MySQL you need to create a database named "peewee_test". To test the Postgres extension module, you will also want to install the HStore extension in the postgres test database:

```
-- install the hstore extension on the peewee_test postgres db.
CREATE EXTENSION hstore;
```

1.1.3 Optional dependencies

Note: To use Peewee, you typically won't need anything outside the standard library, since most Python distributions are compiled with SQLite support. You can test by running `import sqlite3` in the Python console. If you wish to use another database, there are many DB-API 2.0-compatible drivers out there, such as `pymysql` or `psycopg2` for MySQL and Postgres respectively.

- Cython: used to expose additional functionality when using SQLite and to implement things like search result ranking in a performant manner. Since the generated C files are included with the package distribution, Cython is no longer required to use the C extensions.
- apsw: an optional 3rd-party SQLite binding offering greater performance and comprehensive support for SQLite's C APIs. Use with *`APSWDatabase`*.
- gevent is an optional dependency for `SqliteQueueDatabase` (though it works with `threading` just fine).
- BerkeleyDB can be compiled with a SQLite frontend, which works with Peewee. Compiling can be tricky so here are instructions.
- Lastly, if you use the *Flask* framework, there are helper extension modules available.

1.1.4 Note on the SQLite extensions

Peewee includes two SQLite-specific C extensions which provide additional functionality and improved performance for SQLite database users. Peewee will attempt to determine ahead-of-time if SQLite3 is installed, and only build the SQLite extensions if the SQLite shared-library is available on your system.

If, however, you receive errors like the following when attempting to install Peewee, you can explicitly disable the compilation of the SQLite C extensions by settings the `NO_SQLITE` environment variable.

```
fatal error: sqlite3.h: No such file or directory
```

Here is how to install Peewee with the SQLite extensions explicitly disabled:

```
$ NO_SQLITE=1 python setup.py install
```

1.2 Quickstart

This document presents a brief, high-level overview of Peewee's primary features. This guide will cover:

- *Model Definition*
- *Storing data*
- *Retrieving Data*

Note: If you'd like something a bit more meaty, there is a thorough tutorial on *creating a "twitter"-style web app* using peewee and the Flask framework. In the projects `examples/` folder you can find more self-contained Peewee examples, like a blog app.

I **strongly** recommend opening an interactive shell session and running the code. That way you can get a feel for typing in queries.

1.2.1 Model Definition

Model classes, fields and model instances all map to database concepts:

Object	Corresponds to...
Model class	Database table
Field instance	Column on a table
Model instance	Row in a database table

When starting a project with peewee, it's typically best to begin with your data model, by defining one or more *Model* classes:

```
from peewee import *

db = SqliteDatabase('people.db')

class Person(Model):
    name = CharField()
    birthday = DateField()

    class Meta:
        database = db # This model uses the "people.db" database.
```

Note: Peewee will automatically infer the database table name from the name of the class. You can override the default name by specifying a `table_name` attribute in the inner "Meta" class (alongside the `database` attribute). To learn more about how Peewee generates table names, refer to the *Table Names* section.

Also note that we named our model `Person` instead of `People`. This is a convention you should follow – even though the table will contain multiple people, we always name the class using the singular form.

There are lots of *field types* suitable for storing various types of data. Peewee handles converting between *pythonic* values those used by the database, so you can use Python types in your code without having to worry.

Things get interesting when we set up relationships between models using *foreign key relationships*. This is simple with peewee:

```
class Pet(Model):
    owner = ForeignKeyField(Person, backref='pets')
    name = CharField()
    animal_type = CharField()

    class Meta:
        database = db # this model uses the "people.db" database
```

Now that we have our models, let's connect to the database. Although it's not necessary to open the connection explicitly, it is good practice since it will reveal any errors with your database connection immediately, as opposed to some arbitrary time later when the first query is executed. It is also good to close the connection when you are done – for instance, a web app might open a connection when it receives a request, and close the connection when it sends the response.

```
db.connect()
```

We'll begin by creating the tables in the database that will store our data. This will create the tables with the appropriate columns, indexes, sequences, and foreign key constraints:

```
db.create_tables([Person, Pet])
```

1.2.2 Storing data

Let's begin by populating the database with some people. We will use the *save()* and *create()* methods to add and update people's records.

```
from datetime import date
uncle_bob = Person(name='Bob', birthday=date(1960, 1, 15))
uncle_bob.save() # bob is now stored in the database
# Returns: 1
```

Note: When you call *save()*, the number of rows modified is returned.

You can also add a person by calling the *create()* method, which returns a model instance:

```
grandma = Person.create(name='Grandma', birthday=date(1935, 3, 1))
herb = Person.create(name='Herb', birthday=date(1950, 5, 5))
```

To update a row, modify the model instance and call *save()* to persist the changes. Here we will change Grandma's name and then save the changes in the database:

```
grandma.name = 'Grandma L.'
grandma.save()  # Update grandma's name in the database.
# Returns: 1
```

Now we have stored 3 people in the database. Let's give them some pets. Grandma doesn't like animals in the house, so she won't have any, but Herb is an animal lover:

```
bob_kitty = Pet.create(owner=uncle_bob, name='Kitty', animal_type='cat')
herb_fido = Pet.create(owner=herb, name='Fido', animal_type='dog')
herb_mittens = Pet.create(owner=herb, name='Mittens', animal_type='cat')
herb_mittens_jr = Pet.create(owner=herb, name='Mittens Jr', animal_type='cat')
```

After a long full life, Mittens sickens and dies. We need to remove him from the database:

```
herb_mittens.delete_instance() # he had a great life
# Returns: 1
```

Note: The return value of *delete_instance()* is the number of rows removed from the database.

Uncle Bob decides that too many animals have been dying at Herb's house, so he adopts Fido:

```
herb_fido.owner = uncle_bob
herb_fido.save()
```

1.2.3 Retrieving Data

The real strength of our database is in how it allows us to retrieve data through *queries*. Relational databases are excellent for making ad-hoc queries.

Getting single records

Let's retrieve Grandma's record from the database. To get a single record from the database, use `Select.get()`:

```
grandma = Person.select().where(Person.name == 'Grandma L.').get()
```

We can also use the equivalent shorthand *Model.get()*:

```
grandma = Person.get(Person.name == 'Grandma L.')
```

Lists of records

Let's list all the people in the database:

```
for person in Person.select():
    print(person.name)

# prints:
# Bob
# Grandma L.
# Herb
```

Let's list all the cats and their owner's name:

```
query = Pet.select().where(Pet.animal_type == 'cat')
for pet in query:
    print(pet.name, pet.owner.name)

# prints:
```

(continues on next page)

(continued from previous page)

```
# Kitty Bob
# Mittens Jr Herb
```

Attention: There is a big problem with the previous query: because we are accessing `pet.owner.name` and we did not select this relation in our original query, peewee will have to perform an additional query to retrieve the pet's owner. This behavior is referred to as *N+1* and it should generally be avoided.

For an in-depth guide to working with relationships and joins, refer to the *Relationships and Joins* documentation.

We can avoid the extra queries by selecting both *Pet* and *Person*, and adding a *join*.

```
query = (Pet
         .select(Pet, Person)
         .join(Person)
         .where(Pet.animal_type == 'cat'))

for pet in query:
    print(pet.name, pet.owner.name)

# prints:
# Kitty Bob
# Mittens Jr Herb
```

Let's get all the pets owned by Bob:

```
for pet in Pet.select().join(Person).where(Person.name == 'Bob'):
    print(pet.name)

# prints:
# Kitty
# Fido
```

We can do another cool thing here to get bob's pets. Since we already have an object to represent Bob, we can do this instead:

```
for pet in Pet.select().where(Pet.owner == uncle_bob):
    print(pet.name)
```

Sorting

Let's make sure these are sorted alphabetically by adding an `order_by()` clause:

```
for pet in Pet.select().where(Pet.owner == uncle_bob).order_by(Pet.name):
    print(pet.name)

# prints:
# Fido
# Kitty
```

Let's list all the people now, youngest to oldest:

```
for person in Person.select().order_by(Person.birthday.desc()):
    print(person.name, person.birthday)
```

(continues on next page)

(continued from previous page)

```
# prints:
# Bob 1960-01-15
# Herb 1950-05-05
# Grandma L. 1935-03-01
```

Combining filter expressions

Peewee supports arbitrarily-nested expressions. Let's get all the people whose birthday was either:

- before 1940 (grandma)
- after 1959 (bob)

```
d1940 = date(1940, 1, 1)
d1960 = date(1960, 1, 1)
query = (Person
         .select()
         .where((Person.birthday < d1940) | (Person.birthday > d1960)))

for person in query:
    print(person.name, person.birthday)

# prints:
# Bob 1960-01-15
# Grandma L. 1935-03-01
```

Now let's do the opposite. People whose birthday is between 1940 and 1960:

```
query = (Person
         .select()
         .where(Person.birthday.between(d1940, d1960)))

for person in query:
    print(person.name, person.birthday)

# prints:
# Herb 1950-05-05
```

Aggregates and Prefetch

Now let's list all the people *and* how many pets they have:

```
for person in Person.select():
    print(person.name, person.pets.count(), 'pets')

# prints:
# Bob 2 pets
# Grandma L. 0 pets
# Herb 1 pets
```

Once again we've run into a classic example of *N+1* query behavior. In this case, we're executing an additional query for every `Person` returned by the original `SELECT`! We can avoid this by performing a *JOIN* and using a SQL function to aggregate the results.

```
query = (Person
         .select(Person, fn.COUNT(Pet.id).alias('pet_count'))
         .join(Pet, JOIN.LEFT_OUTER)  # include people without pets.
         .group_by(Person)
         .order_by(Person.name))

for person in query:
    # "pet_count" becomes an attribute on the returned model instances.
    print(person.name, person.pet_count, 'pets')

# prints:
# Bob 2 pets
# Grandma L. 0 pets
# Herb 1 pets
```

Note: Peewee provides a magical helper *fn()*, which can be used to call any SQL function. In the above example, `fn.COUNT(Pet.id).alias('pet_count')` would be translated into `COUNT(pet.id) AS pet_count`.

Now let's list all the people and the names of all their pets. As you may have guessed, this could easily turn into another *N+1* situation if we're not careful.

Before diving into the code, consider how this example is different from the earlier example where we listed all the pets and their owner's name. A pet can only have one owner, so when we performed the join from `Pet` to `Person`, there was always going to be a single match. The situation is different when we are joining from `Person` to `Pet` because a person may have zero pets or they may have several pets. Because we're using a relational databases, if we were to do a join from `Person` to `Pet` then every person with multiple pets would be repeated, once for each pet.

It would look like this:

```
query = (Person
         .select(Person, Pet)
         .join(Pet, JOIN.LEFT_OUTER)
         .order_by(Person.name, Pet.name))
for person in query:
    # We need to check if they have a pet instance attached, since not all
    # people have pets.
    if hasattr(person, 'pet'):
        print(person.name, person.pet.name)
    else:
        print(person.name, 'no pets')

# prints:
# Bob Fido
# Bob Kitty
# Grandma L. no pets
# Herb Mittens Jr
```

Usually this type of duplication is undesirable. To accommodate the more common (and intuitive) workflow of listing a person and attaching **a list** of that person's pets, we can use a special method called *prefetch()*:

```
query = Person.select().order_by(Person.name).prefetch(Pet)
for person in query:
    print(person.name)
    for pet in person.pets:
        print('  *', pet.name)
```

(continues on next page)

(continued from previous page)

```
# prints:
# Bob
#   * Kitty
#   * Fido
# Grandma L.
# Herb
#   * Mittens Jr
```

SQL Functions

One last query. This will use a SQL function to find all people whose names start with either an upper or lower-case *G*:

```
expression = fn.Lower(fn.Substr(Person.name, 1, 1)) == 'g'
for person in Person.select().where(expression):
    print(person.name)

# prints:
# Grandma L.
```

This is just the basics! You can make your queries as complex as you like. Check the documentation on *Querying* for more info.

1.2.4 Database

We're done with our database, let's close the connection:

```
db.close()
```

In an actual application, there are some established patterns for how you would manage your database connection lifetime. For example, a web application will typically open a connection at start of request, and close the connection after generating the response. A *connection pool* can help eliminate latency associated with startup costs.

To learn about setting up your database, see the *Database* documentation, which provides many examples. Peewee also supports *configuring the database at run-time* as well as setting or changing the database at any time.

Working with existing databases

If you already have a database, you can autogenerate peewee models using *pwiz, a model generator*. For instance, if I have a postgresql database named *charles_blog*, I might run:

```
python -m pwiz -e postgresql charles_blog > blog_models.py
```

1.2.5 What next?

That's it for the quickstart. If you want to look at a full web-app, check out the *Example app*.

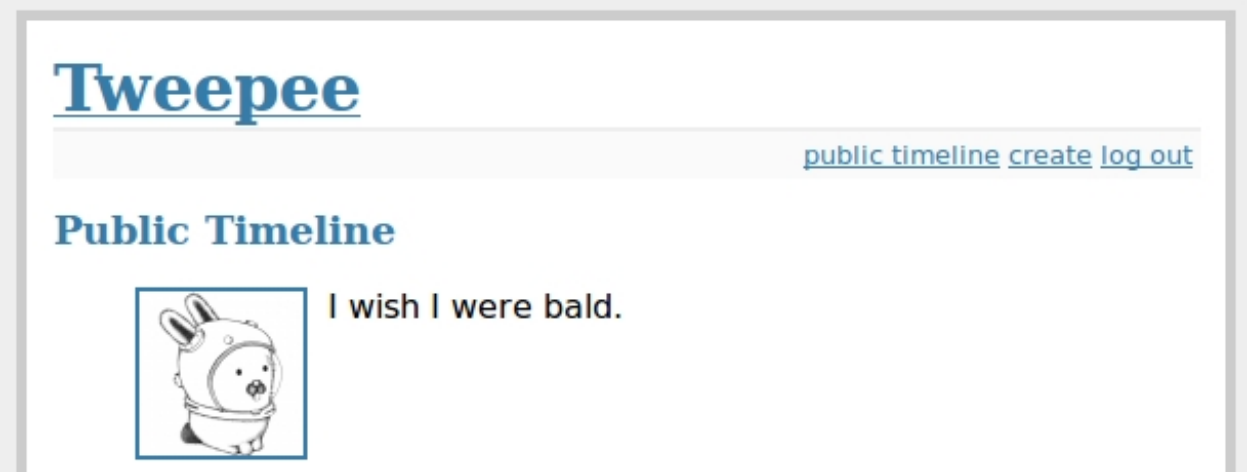
1.3 Example app

We'll be building a simple *twitter*-like site. The source code for the example can be found in the `examples/twitter` directory. You can also browse the source-code on github. There is also an example blog app if that's more to your liking, however it is not covered in this guide.

The example app uses the flask web framework which is very easy to get started with. If you don't have flask already, you will need to install it to run the example:

```
pip install flask
```

1.3.1 Running the example



After ensuring that flask is installed, `cd` into the twitter example directory and execute the `run_example.py` script:

```
python run_example.py
```

The example app will be accessible at http://localhost:5000/

1.3.2 Diving into the code

For simplicity all example code is contained within a single module, `examples/twitter/app.py`. For a guide on structuring larger Flask apps with peewee, check out Structuring Flask Apps.

Models

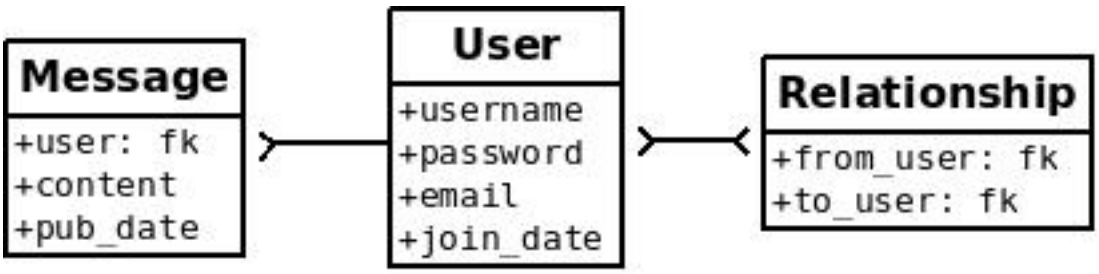
In the spirit of the popular web framework Django, peewee uses declarative model definitions. If you're not familiar with Django, the idea is that you declare a model class for each table. The model class then defines one or more field attributes which correspond to the table's columns. For the twitter clone, there are just three models:

***User*:** Represents a user account and stores the username and password, an email address for generating avatars using *gravatar*, and a datetime field indicating when that account was created.

***Relationship*:** This is a utility model that contains two foreign-keys to the *User* model and stores which users follow one another.

Message: Analogous to a tweet. The Message model stores the text content of the tweet, when it was created, and who posted it (foreign key to User).

If you like UML, these are the tables and relationships:



In order to create these models we need to instantiate a *SqliteDatabase* object. Then we define our model classes, specifying the columns as *Field* instances on the class.

```
# create a peewee database instance -- our models will use this database to
# persist information
database = SqliteDatabase(DATABASE)

# model definitions -- the standard "pattern" is to define a base model class
# that specifies which database to use.  then, any subclasses will automatically
# use the correct storage.
class BaseModel(Model):
    class Meta:
        database = database

# the user model specifies its fields (or columns) declaratively, like django
class User(BaseModel):
    username = CharField(unique=True)
    password = CharField()
    email = CharField()
    join_date = DateTimeField()

# this model contains two foreign keys to user -- it essentially allows us to
# model a "many-to-many" relationship between users.  by querying and joining
# on different columns we can expose who a user is "related to" and who is
# "related to" a given user
class Relationship(BaseModel):
    from_user = ForeignKeyField(User, backref='relationships')
    to_user = ForeignKeyField(User, backref='related_to')

    class Meta:
        # `indexes` is a tuple of 2-tuples, where the 2-tuples are
        # a tuple of column names to index and a boolean indicating
        # whether the index is unique or not.
        indexes = (
            # Specify a unique multi-column index on from/to-user.
            (('from_user', 'to_user'), True),
        )

# a dead simple one-to-many relationship: one user has 0..n messages, exposed by
# the foreign key.  because we didn't specify, a users messages will be accessible
# as a special attribute, User.messages
class Message(BaseModel):
    user = ForeignKeyField(User, backref='messages')
    content = TextField()
```

(continues on next page)

(continued from previous page)

```
    pub_date = DateTimeField()
```

Note: Note that we create a *BaseModel* class that simply defines what database we would like to use. All other models then extend this class and will also use the correct database connection.

Peewee supports many different *field types* which map to different column types commonly supported by database engines. Conversion between python types and those used in the database is handled transparently, allowing you to use the following in your application:

- Strings (unicode or otherwise)
- Integers, floats, and `Decimal` numbers.
- Boolean values
- Dates, times and datetimes
- `None` (NULL)
- Binary data

Creating tables

In order to start using the models, its necessary to create the tables. This is a one-time operation and can be done quickly using the interactive interpreter. We can create a small helper function to accomplish this:

```
def create_tables():
    with database:
        database.create_tables([User, Relationship, Message])
```

Open a python shell in the directory alongside the example app and execute the following:

```
>>> from app import *
>>> create_tables()
```

Note: If you encounter an *ImportError* it means that either *flask* or *peewee* was not found and may not be installed correctly. Check the *Installing and Testing* document for instructions on installing peewee.

Every model has a *`create_table()`* classmethod which runs a SQL *CREATE TABLE* statement in the database. This method will create the table, including all columns, foreign-key constraints, indexes, and sequences. Usually this is something you'll only do once, whenever a new model is added.

Peewee provides a helper method *`Database.create_tables()`* which will resolve inter-model dependencies and call *`create_table()`* on each model, ensuring the tables are created in order.

Note: Adding fields after the table has been created will require you to either drop the table and re-create it or manually add the columns using an *ALTER TABLE* query.

Alternatively, you can use the *schema migrations* extension to alter your database schema using Python.

Establishing a database connection

You may have noticed in the above model code that there is a class defined on the base model named *Meta* that sets the `database` attribute. Peewee allows every model to specify which database it uses. There are many *Meta options* you can specify which control the behavior of your model.

This is a peewee idiom:

```
DATABASE = 'tweepee.db'

# Create a database instance that will manage the connection and
# execute queries
database = SqliteDatabase(DATABASE)

# Create a base-class all our models will inherit, which defines
# the database we'll be using.
class BaseModel(Model):
    class Meta:
        database = database
```

When developing a web application, it's common to open a connection when a request starts, and close it when the response is returned. **You should always manage your connections explicitly**. For instance, if you are using a *connection pool*, connections will only be recycled correctly if you call *`connect()`* and *`close()`*.

We will tell flask that during the request/response cycle we need to create a connection to the database. Flask provides some handy decorators to make this a snap:

```
@app.before_request
def before_request():
    database.connect()

@app.after_request
def after_request(response):
    database.close()
    return response
```

Note: Peewee uses thread local storage to manage connection state, so this pattern can be used with multi-threaded WSGI servers.

Making queries

In the *User* model there are a few instance methods that encapsulate some user-specific functionality:

- `following()`: who is this user following?
- `followers()`: who is following this user?

These methods are similar in their implementation but with an important difference in the SQL *JOIN* and *WHERE* clauses:

```
def following(self):
    # query other users through the "relationship" table
    return (User
            .select()
            .join(Relationship, on=Relationship.to_user)
            .where(Relationship.from_user == self)
```

(continues on next page)

(continued from previous page)

```
            .order_by(User.username))

def followers(self):
    return (User
            .select()
            .join(Relationship, on=Relationship.from_user)
            .where(Relationship.to_user == self)
            .order_by(User.username))
```

Creating new objects

When a new user wants to join the site we need to make sure the username is available, and if so, create a new *User* record. Looking at the *join()* view, we can see that our application attempts to create the User using *Model.create()*. We defined the *User.username* field with a unique constraint, so if the username is taken the database will raise an `IntegrityError`.

```
try:
    with database.atomic():
        # Attempt to create the user. If the username is taken, due to the
        # unique constraint, the database will raise an IntegrityError.
        user = User.create(
            username=request.form['username'],
            password=md5(request.form['password']).hexdigest(),
            email=request.form['email'],
            join_date=datetime.datetime.now())

    # mark the user as being 'authenticated' by setting the session vars
    auth_user(user)
    return redirect(url_for('homepage'))

except IntegrityError:
    flash('That username is already taken')
```

We will use a similar approach when a user wishes to follow someone. To indicate a following relationship, we create a row in the *Relationship* table pointing from one user to another. Due to the unique index on `from_user` and `to_user`, we will be sure not to end up with duplicate rows:

```
user = get_object_or_404(User, username=username)
try:
    with database.atomic():
        Relationship.create(
            from_user=get_current_user(),
            to_user=user)
except IntegrityError:
    pass
```

Performing subqueries

If you are logged-in and visit the twitter homepage, you will see tweets from the users that you follow. In order to implement this cleanly, we can use a subquery:

Note: The subquery, `user.following()`, by default would ordinarily select all the columns on the `User` model.

Because we're using it as a subquery, peewee will only select the primary key.

```
# python code
user = get_current_user()
messages = (Message
            .select()
            .where(Message.user.in_(user.following()))
            .order_by(Message.pub_date.desc()))
```

This code corresponds to the following SQL query:

```
SELECT t1."id", t1."user_id", t1."content", t1."pub_date"
FROM "message" AS t1
WHERE t1."user_id" IN (
    SELECT t2."id"
    FROM "user" AS t2
    INNER JOIN "relationship" AS t3
        ON t2."id" = t3."to_user_id"
    WHERE t3."from_user_id" = ?
)
```

Other topics of interest

There are a couple other neat things going on in the example app that are worth mentioning briefly.

- Support for paginating lists of results is implemented in a simple function called `object_list` (after it's corollary in Django). This function is used by all the views that return lists of objects.

```
def object_list(template_name, qr, var_name='object_list', **kwargs):
    kwargs.update(
        page=int(request.args.get('page', 1)),
        pages=qr.count() / 20 + 1)
    kwargs[var_name] = qr.paginate(kwargs['page'])
    return render_template(template_name, **kwargs)
```

- Simple authentication system with a `login_required` decorator. The first function simply adds user data into the current session when a user successfully logs in. The decorator `login_required` can be used to wrap view functions, checking for whether the session is authenticated and if not redirecting to the login page.

```
def auth_user(user):
    session['logged_in'] = True
    session['user'] = user
    session['username'] = user.username
    flash('You are logged in as %s' % (user.username))

def login_required(f):
    @wraps(f)
    def inner(*args, **kwargs):
        if not session.get('logged_in'):
            return redirect(url_for('login'))
        return f(*args, **kwargs)
    return inner
```

- Return a 404 response instead of throwing exceptions when an object is not found in the database.

```
def get_object_or_404(model, *expressions):
    try:
        return model.get(*expressions)
    except model.DoesNotExist:
        abort(404)
```

Note: To avoid having to frequently copy/paste *object_list()* or *get_object_or_404()*, these functions are included as part of the playhouse *flask extension module*.

```
from playhouse.flask_utils import get_object_or_404, object_list
```

1.3.3 More examples

There are more examples included in the peewee examples directory, including:

- Example blog app using Flask and peewee. Also see accompanying blog post.
- An encrypted command-line diary. There is a companion blog post you might enjoy as well.
- Analytics web-service (like a lite version of Google Analytics). Also check out the companion blog post.

Note: Like these snippets and interested in more? Check out flask-peewee - a flask plugin that provides a django-like Admin interface, RESTful API, Authentication and more for your peewee models.

1.4 Using Peewee Interactively

Peewee contains helpers for working interactively from a Python interpreter or something like a Jupyter notebook. For this example, we'll assume that we have a pre-existing Sqlite database with the following simple schema:

```
CREATE TABLE IF NOT EXISTS "event" (
    "id" INTEGER NOT NULL PRIMARY KEY,
    "key" TEXT NOT NULL,
    "timestamp" DATETIME NOT NULL,
    "metadata" TEXT NOT NULL);
```

To experiment with querying this database from an interactive interpreter session, we would start our interpreter and import the following helpers:

- `peewee.SqliteDatabase` - to reference the "events.db"
- `playhouse.reflection.generate_models` - to generate models from an existing database.
- `playhouse.reflection.print_model` - to view the model definition.
- `playhouse.reflection.print_table_sql` - to view the table SQL.

Our terminal session might look like this:

```
>>> from peewee import SqliteDatabase
>>> from playhouse.reflection import generate_models, print_model, print_table_sql
>>>
```

The *generate_models()* function will introspect the database and generate model classes for all the tables that are found. This is a handy way to get started and can save a lot of typing. The function returns a dictionary keyed by the table name, with the generated model as the corresponding value:

```
>>> db = SqliteDatabase('events.db')
>>> models = generate_models(db)
>>> list(models.items())
[('events', <Model: event>)]

>>> globals().update(models)  # Inject models into global namespace.
>>> event
<Model: event>
```

To take a look at the model definition, which lists the model's fields and data-type, we can use the *print_model()* function:

```
>>> print_model(event)
event
  id AUTO
  key TEXT
  timestamp DATETIME
  metadata TEXT
```

We can also generate a SQL `CREATE TABLE` for the introspected model, if you find that easier to read. This should match the actual table definition in the introspected database:

```
>>> print_table_sql(event)
CREATE TABLE IF NOT EXISTS "event" (
  "id" INTEGER NOT NULL PRIMARY KEY,
  "key" TEXT NOT NULL,
  "timestamp" DATETIME NOT NULL,
  "metadata" TEXT NOT NULL)
```

Now that we are familiar with the structure of the table we're working with, we can run some queries on the generated `event` model:

```
>>> for e in event.select().order_by(event.timestamp).limit(5):
...     print(e.key, e.timestamp)
...
e00 2019-01-01 00:01:00
e01 2019-01-01 00:02:00
e02 2019-01-01 00:03:00
e03 2019-01-01 00:04:00
e04 2019-01-01 00:05:00

>>> event.select(fn.MIN(event.timestamp), fn.MAX(event.timestamp)).scalar(as_
↪tuple=True)
(datetime.datetime(2019, 1, 1, 0, 1), datetime.datetime(2019, 1, 1, 1, 0))

>>> event.select().count()  # Or, len(event)
60
```

For more information about these APIs and other similar reflection utilities, see the *Reflection* section of the *playhouse extensions* document.

To generate an actual Python module containing model definitions for an existing database, you can use the command-line *pwiz* tool. Here is a quick example:

```
$ pwiz -e sqlite events.db > events.py
```

The `events.py` file will now be an import-able module containing a database instance (referencing the `events.db`) along with model definitions for any tables found in the database. `pwiz` does some additional nice things like introspecting indexes and adding proper flags for `NULL`/`NOT NULL` constraints, etc.

The APIs discussed in this section:

- *generate_models()*
- *print_model()*
- *print_table_sql()*

More low-level APIs are also available on the *Database* instance:

- *Database.get_tables()*
- *Database.get_indexes()*
- *Database.get_columns()* (for a given table)
- *Database.get_primary_keys()* (for a given table)
- *Database.get_foreign_keys()* (for a given table)

1.5 Contributing

In order to continually improve, Peewee needs the help of developers like you. Whether it's contributing patches, submitting bug reports, or just asking and answering questions, you are helping to make Peewee a better library.

In this document I'll describe some of the ways you can help.

1.5.1 Patches

Do you have an idea for a new feature, or is there a clunky API you'd like to improve? Before coding it up and submitting a pull-request, open a new issue on GitHub describing your proposed changes. This doesn't have to be anything formal, just a description of what you'd like to do and why.

When you're ready, you can submit a pull-request with your changes. Successful patches will have the following:

- Unit tests.
- Documentation, both prose form and general *API documentation*.
- Code that conforms stylistically with the rest of the Peewee codebase.

1.5.2 Bugs

If you've found a bug, please check to see if it has already been reported, and if not create an issue on GitHub. The more information you include, the more quickly the bug will get fixed, so please try to include the following:

- Traceback and the error message (please format your code!)
- Relevant portions of your code or code to reproduce the error
- Peewee version: `python -c "from peewee import __version__; print(__version__)"`
- Which database you're using

If you have found a bug in the code and submit a failing test-case, then hats-off to you, you are a hero!

1.5.3 Questions

If you have questions about how to do something with peewee, then I recommend either:

- Ask on StackOverflow. I check SO just about every day for new peewee questions and try to answer them. This has the benefit also of preserving the question and answer for other people to find.
- Ask on the mailing list, https://groups.google.com/group/peewee-orm

1.6 Database

The Peewee *Database* object represents a connection to a database. The *Database* class is instantiated with all the information needed to open a connection to a database, and then can be used to:

- Open and close connections.
- Execute queries.
- Manage transactions (and savepoints).
- Introspect tables, columns, indexes, and constraints.

Peewee comes with support for SQLite, MySQL and Postgres. Each database class provides some basic, database-specific configuration options.

```
from peewee import *

# SQLite database using WAL journal mode and 64MB cache.
sqlite_db = SqliteDatabase('/path/to/app.db', pragmas={
    'journal_mode': 'wal',
    'cache_size': -1024 * 64})

# Connect to a MySQL database on network.
mysql_db = MySQLDatabase('my_app', user='app', password='db_password',
                         host='10.1.0.8', port=3306)

# Connect to a Postgres database.
pg_db = PostgresqlDatabase('my_app', user='postgres', password='secret',
                           host='10.1.0.9', port=5432)
```

Peewee provides advanced support for SQLite, Postgres and CockroachDB via database-specific extension modules. To use the extended-functionality, import the appropriate database-specific module and use the database class provided:

```
from playhouse.sqlite_ext import SqliteExtDatabase

# Use SQLite (will register a REGEXP function and set busy timeout to 3s).
db = SqliteExtDatabase('/path/to/app.db', regexp_function=True, timeout=3,
                       pragmas={'journal_mode': 'wal'})


from playhouse.postgres_ext import PostgresqlExtDatabase

# Use Postgres (and register hstore extension).
db = PostgresqlExtDatabase('my_app', user='postgres', register_hstore=True)
```

(continues on next page)

(continued from previous page)

```
from playhouse.cockroachdb import CockroachDatabase

# Use CockroachDB.
db = CockroachDatabase('my_app', user='root', port=26257, host='10.1.0.8')
```

For more information on database extensions, see:

- *Postgresql Extensions*
- *SQLite Extensions*
- *Cockroach Database*
- *Sqlcipher backend* (encrypted SQLite database).
- *apsw, an advanced sqlite driver*
- *SqliteQ*

1.6.1 Initializing a Database

The `Database` initialization method expects the name of the database as the first parameter. Subsequent keyword arguments are passed to the underlying database driver when establishing the connection, allowing you to pass vendor-specific parameters easily.

For instance, with Postgresql it is common to need to specify the `host`, `user` and `password` when creating your connection. These are not standard Peewee `Database` parameters, so they will be passed directly back to `psycopg2` when creating connections:

```
db = PostgresqlDatabase(
    'database_name',  # Required by Peewee.
    user='postgres',  # Will be passed directly to psycopg2.
    password='secret',  # Ditto.
    host='db.mysite.com')  # Ditto.
```

As another example, the `pymysql` driver accepts a `charset` parameter which is not a standard Peewee `Database` parameter. To set this value, simply pass in `charset` alongside your other values:

```
db = MySQLDatabase('database_name', user='www-data', charset='utf8mb4')
```

Consult your database driver's documentation for the available parameters:

- Postgres: psycopg2
- MySQL: MySQLdb
- MySQL: pymysql
- SQLite: sqlite3
- CockroachDB: see psycopg2

1.6.2 Using Postgresql

To connect to a Postgresql database, we will use `PostgresqlDatabase`. The first parameter is always the name of the database, and after that you can specify arbitrary psycopg2 parameters.

```
psql_db = PostgresqlDatabase('my_database', user='postgres')

class BaseModel(Model):
    """A base model that will use our Postgresql database"""
    class Meta:
        database = psql_db

class User(BaseModel):
    username = CharField()
```

The *Playhouse, extensions to Peewee* contains a *Postgresql extension module* which provides many postgres-specific features such as:

- *Arrays*
- *HStore*
- *JSON*
- *Server-side cursors*
- And more!

If you would like to use these awesome features, use the *PostgresqlExtDatabase* from the `playhouse.postgres_ext` module:

```
from playhouse.postgres_ext import PostgresqlExtDatabase

psql_db = PostgresqlExtDatabase('my_database', user='postgres')
```

Isolation level

As of Peewee 3.9.7, the isolation level can be specified as an initialization parameter, using the symbolic constants in `psycopg2.extensions`:

```
from psycopg2.extensions import ISOLATION_LEVEL_SERIALIZABLE

db = PostgresqlDatabase('my_app', user='postgres', host='db-host',
                        isolation_level=ISOLATION_LEVEL_SERIALIZABLE)
```

Note: In older versions, you can manually set the isolation level on the underlying psycopg2 connection. This can be done in a one-off fashion:

```
db = PostgresqlDatabase(...)
conn = db.connection()  # returns current connection.

from psycopg2.extensions import ISOLATION_LEVEL_SERIALIZABLE
conn.set_isolation_level(ISOLATION_LEVEL_SERIALIZABLE)
```

To run this every time a connection is created, subclass and implement the `_initialize_database()` hook, which is designed for this purpose:

```
class SerializedPostgresqlDatabase(PostgresqlDatabase):
    def _initialize_connection(self, conn):
        conn.set_isolation_level(ISOLATION_LEVEL_SERIALIZABLE)
```

1.6.3 Using CockroachDB

Connect to CockroachDB (CRDB) using the *CockroachDatabase* database class, defined in `playhouse.cockroachdb`:

```
from playhouse.cockroachdb import CockroachDatabase

db = CockroachDatabase('my_app', user='root', port=26257, host='localhost')
```

CRDB provides client-side transaction retries, which are available using a special *CockroachDatabase.run_transaction()* helper-method. This method accepts a callable, which is responsible for executing any transactional statements that may need to be retried.

Simplest possible example of *run_transaction()*:

```
def create_user(email):
    # Callable that accepts a single argument (the database instance) and
    # which is responsible for executing the transactional SQL.
    def callback(db_ref):
        return User.create(email=email)

    return db.run_transaction(callback, max_attempts=10)

huey = create_user('huey@example.com')
```

Note: The `cockroachdb.ExceededMaxAttempts` exception will be raised if the transaction cannot be committed after the given number of attempts. If the SQL is mal-formed, violates a constraint, etc., then the function will raise the exception to the caller.

For more information, see:

- *CRDB extension documentation*
- *Arrays* (postgres-specific, but applies to CRDB)
- *JSON* (postgres-specific, but applies to CRDB)

1.6.4 Using SQLite

To connect to a SQLite database, we will use *SqliteDatabase*. The first parameter is the filename containing the database, or the string `':memory:'` to create an in-memory database. After the database filename, you can specify a list or pragmas or any other arbitrary sqlite3 parameters.

```
sqlite_db = SqliteDatabase('my_app.db', pragmas={'journal_mode': 'wal'})

class BaseModel(Model):
    """A base model that will use our Sqlite database."""
    class Meta:
        database = sqlite_db

class User(BaseModel):
    username = TextField()
    # etc, etc
```

Peewee includes a *SQLite extension module* which provides many SQLite-specific features such as *full-text search*, *json extension support*, and much, much more. If you would like to use these awesome features, use the *SqliteExtDatabase* from the `playhouse.sqlite_ext` module:

```
from playhouse.sqlite_ext import SqliteExtDatabase

sqlite_db = SqliteExtDatabase('my_app.db', pragmas={
    'journal_mode': 'wal',  # WAL-mode.
    'cache_size': -64 * 1000,  # 64MB cache.
    'synchronous': 0})  # Let the OS manage syncing.
```

PRAGMA statements

SQLite allows run-time configuration of a number of parameters through `PRAGMA` statements (SQLite documentation). These statements are typically run when a new database connection is created. To run one or more `PRAGMA` statements against new connections, you can specify them as a dictionary or a list of 2-tuples containing the pragma name and value:

```
db = SqliteDatabase('my_app.db', pragmas={
    'journal_mode': 'wal',
    'cache_size': 10000,  # 10000 pages, or ~40MB
    'foreign_keys': 1,  # Enforce foreign-key constraints
})
```

PRAGMAs may also be configured dynamically using either the *pragma()* method or the special properties exposed on the *SqliteDatabase* object:

```
# Set cache size to 64MB for *current connection*.
db.pragma('cache_size', -1024 * 64)

# Same as above.
db.cache_size = -1024 * 64

# Read the value of several pragmas:
print('cache_size:', db.cache_size)
print('foreign_keys:', db.foreign_keys)
print('journal_mode:', db.journal_mode)
print('page_size:', db.page_size)

# Set foreign_keys pragma on current connection *AND* on all
# connections opened subsequently.
db.pragma('foreign_keys', 1, permanent=True)
```

Attention: Pragmas set using the *pragma()* method, by default, do not persist after the connection is closed. To configure a pragma to be run whenever a connection is opened, specify `permanent=True`.

Note: A full list of PRAGMA settings, their meaning and accepted values can be found in the SQLite documentation: http://sqlite.org/pragma.html

Recommended Settings

The following settings are what I use with SQLite for a typical web application database.

pragma	recommended setting	explanation
journal_mode	wal	allow readers and writers to co-exist
cache_size	-1 * data_size_kb	set page-cache size in KiB, e.g. -32000 = 32MB
foreign_keys	1	enforce foreign-key constraints
ignore_check_constraints	0	enforce CHECK constraints
synchronous	0	let OS handle fsync (use with caution)

Example database using the above options:

```
db = SqliteDatabase('my_app.db', pragmas={
    'journal_mode': 'wal',
    'cache_size': -1 * 64000,  # 64MB
    'foreign_keys': 1,
    'ignore_check_constraints': 0,
    'synchronous': 0})
```

User-defined functions

SQLite can be extended with user-defined Python code. The *SqliteDatabase* class supports three types of user-defined extensions:

- Functions - which take any number of parameters and return a single value.
- Aggregates - which aggregate parameters from multiple rows and return a single value.
- Collations - which describe how to sort some value.

Note: For even more extension support, see *SqliteExtDatabase*, which is in the `playhouse.sqlite_ext` module.

Example user-defined function:

```
db = SqliteDatabase('analytics.db')

from urllib.parse import urlparse

@db.func('hostname')
def hostname(url):
    if url is not None:
        return urlparse(url).netloc

# Call this function in our code:
# The following finds the most common hostnames of referrers by count:
query = (PageView
         .select(fn.hostname(PageView.referrer), fn.COUNT(PageView.id))
         .group_by(fn.hostname(PageView.referrer))
         .order_by(fn.COUNT(PageView.id).desc()))
```

Example user-defined aggregate:

```
from hashlib import md5

@db.aggregate('md5')
class MD5Checksum(object):
    def __init__(self):
        self.checksum = md5()

    def step(self, value):
        self.checksum.update(value.encode('utf-8'))

    def finalize(self):
        return self.checksum.hexdigest()

# Usage:
# The following computes an aggregate MD5 checksum for files broken
# up into chunks and stored in the database.
query = (FileChunk
         .select(FileChunk.filename, fn.MD5(FileChunk.data))
         .group_by(FileChunk.filename)
         .order_by(FileChunk.filename, FileChunk.sequence))
```

Example collation:

```
@db.collation('ireverse')
def collate_reverse(s1, s2):
    # Case-insensitive reverse.
    s1, s2 = s1.lower(), s2.lower()
    return (s1 < s2) - (s1 > s2)  # Equivalent to -cmp(s1, s2)

# To use this collation to sort books in reverse order...
Book.select().order_by(collate_reverse.collation(Book.title))

# Or...
Book.select().order_by(Book.title.asc(collation='reverse'))
```

Example user-defined table-value function (see *TableFunction* and *table_function*) for additional details:

```
from playhouse.sqlite_ext import TableFunction

db = SqliteDatabase('my_app.db')

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

    def initialize(self, start=0, stop=None, step=1):
        """
        Table-functions declare an initialize() method, which is
        called with whatever arguments the user has called the
        function with.
        """
        self.start = self.current = start
        self.stop = stop or float('Inf')
        self.step = step

    def iterate(self, idx):
```

(continues on next page)

(continued from previous page)

```
        """
        Iterate is called repeatedly by the SQLite database engine
        until the required number of rows has been read **or** the
        function raises a `StopIteration` signalling no more rows
        are available.
        """
        if self.current > self.stop:
            raise StopIteration

        ret, self.current = self.current, self.current + self.step
        return (ret,)

# Usage:
cursor = db.execute_sql('SELECT * FROM series(?, ?, ?)', (0, 5, 2))
for value, in cursor:
    print(value)

# Prints:
# 0
# 2
# 4
```

For more information, see:

- *SqliteDatabase.func()*
- *SqliteDatabase.aggregate()*
- *SqliteDatabase.collation()*
- *SqliteDatabase.table_function()*
- For even more SQLite extensions, see *SQLite Extensions*

Set locking mode for transaction

SQLite transactions can be opened in three different modes:

- *Deferred* (**default**) - only acquires lock when a read or write is performed. The first read creates a shared lock and the first write creates a reserved lock. Because the acquisition of the lock is deferred until actually needed, it is possible that another thread or process could create a separate transaction and write to the database after the BEGIN on the current thread has executed.
- *Immediate* - a reserved lock is acquired immediately. In this mode, no other database may write to the database or open an *immediate* or *exclusive* transaction. Other processes can continue to read from the database, however.
- *Exclusive* - opens an exclusive lock which prevents all (except for read uncommitted) connections from accessing the database until the transaction is complete.

Example specifying the locking mode:

```
db = SqliteDatabase('app.db')

with db.atomic('EXCLUSIVE'):
    do_something()


@db.atomic('IMMEDIATE')
```

(continues on next page)

(continued from previous page)

```
def some_other_function():
    # This function is wrapped in an "IMMEDIATE" transaction.
    do_something_else()
```

For more information, see the SQLite locking documentation. To learn more about transactions in Peewee, see the *Managing Transactions* documentation.

APSW, an Advanced SQLite Driver

Peewee also comes with an alternate SQLite database that uses *apsw, an advanced sqlite driver*, an advanced Python SQLite driver. More information on APSW can be obtained on the APSW project website. APSW provides special features like:

- Virtual tables, virtual file-systems, Blob I/O, backups and file control.
- Connections can be shared across threads without any additional locking.
- Transactions are managed explicitly by your code.
- Unicode is handled *correctly*.
- APSW is faster that the standard library sqlite3 module.
- Exposes pretty much the entire SQLite C API to your Python app.

If you would like to use APSW, use the *APSWDatabase* from the *apsw_ext* module:

```
from playhouse.apsw_ext import APSWDatabase

apsw_db = APSWDatabase('my_app.db')
```

1.6.5 Using MySQL

To connect to a MySQL database, we will use *MySQLDatabase*. After the database name, you can specify arbitrary connection parameters that will be passed back to the driver (either MySQLdb or pymysql).

```
mysql_db = MySQLDatabase('my_database')

class BaseModel(Model):
    """A base model that will use our MySQL database"""
    class Meta:
        database = mysql_db

class User(BaseModel):
    username = CharField()
    # etc, etc
```

Error 2006: MySQL server has gone away

This particular error can occur when MySQL kills an idle database connection. This typically happens with web apps that do not explicitly manage database connections. What happens is your application starts, a connection is opened to handle the first query that executes, and, since that connection is never closed, it remains open, waiting for more queries.

To fix this, make sure you are explicitly connecting to the database when you need to execute queries, and close your connection when you are done. In a web-application, this typically means you will open a connection when a request comes in, and close the connection when you return a response.

See the *Framework Integration* section for examples of configuring common web frameworks to manage database connections.

1.6.6 Connecting using a Database URL

The playhouse module *Database URL* provides a helper `connect()` function that accepts a database URL and returns a `Database` instance.

Example code:

```
import os

from peewee import *
from playhouse.db_url import connect

# Connect to the database URL defined in the environment, falling
# back to a local Sqlite database if no database URL is specified.
db = connect(os.environ.get('DATABASE') or 'sqlite:///default.db')

class BaseModel(Model):
    class Meta:
        database = db
```

Example database URLs:

- `sqlite:///my_database.db` will create a `SqliteDatabase` instance for the file `my_database.db` in the current directory.
- `sqlite:///:memory:` will create an in-memory `SqliteDatabase` instance.
- `postgresql://postgres:my_password@localhost:5432/my_database` will create a `PostgresqlDatabase` instance. A username and password are provided, as well as the host and port to connect to.
- `mysql://user:passwd@ip:port/my_db` will create a `MySQLDatabase` instance for the local MySQL database *my_db*.
- *More examples in the db_url documentation*.

1.6.7 Run-time database configuration

Sometimes the database connection settings are not known until run-time, when these values may be loaded from a configuration file or the environment. In these cases, you can *defer* the initialization of the database by specifying `None` as the database_name.

```
database = PostgresqlDatabase(None)  # Un-initialized database.

class SomeModel(Model):
    class Meta:
        database = database
```

If you try to connect or issue any queries while your database is uninitialized you will get an exception:

```
>>> database.connect()
Exception: Error, database not properly initialized before opening connection
```

To initialize your database, call the *init()* method with the database name and any additional keyword arguments:

```
database_name = input('What is the name of the db? ')
database.init(database_name, host='localhost', user='postgres')
```

For even more control over initializing your database, see the next section, *Dynamically defining a database*.

1.6.8 Dynamically defining a database

For even more control over how your database is defined/initialized, you can use the *DatabaseProxy* helper. *DatabaseProxy* objects act as a placeholder, and then at run-time you can swap it out for a different object. In the example below, we will swap out the database depending on how the app is configured:

```
database_proxy = DatabaseProxy()  # Create a proxy for our db.

class BaseModel(Model):
    class Meta:
        database = database_proxy  # Use proxy for our DB.

class User(BaseModel):
    username = CharField()

# Based on configuration, use a different database.
if app.config['DEBUG']:
    database = SqliteDatabase('local.db')
elif app.config['TESTING']:
    database = SqliteDatabase(':memory:')
else:
    database = PostgresqlDatabase('mega_production_db')

# Configure our proxy to use the db we specified in config.
database_proxy.initialize(database)
```

Warning: Only use this method if your actual database driver varies at run-time. For instance, if your tests and local dev environment run on SQLite, but your deployed app uses PostgreSQL, you can use the *DatabaseProxy* to swap out engines at run-time.

However, if it is only connection values that vary at run-time, such as the path to the database file, or the database host, you should instead use *Database.init()*. See *Run-time database configuration* for more details.

Note: It may be easier to avoid the use of *DatabaseProxy* and instead use *Database.bind()* and related methods to set or change the database. See *Setting the database at run-time* for details.

1.6.9 Setting the database at run-time

We have seen three ways that databases can be configured with Peewee:

```
# The usual way:
db = SqliteDatabase('my_app.db', pragmas={'journal_mode': 'wal'})


# Specify the details at run-time:
db = SqliteDatabase(None)
...
db.init(db_filename, pragmas={'journal_mode': 'wal'})


# Or use a placeholder:
db = DatabaseProxy()
...
db.initialize(SqliteDatabase('my_app.db', pragmas={'journal_mode': 'wal'}))
```

Peewee can also set or change the database for your model classes. This technique is used by the Peewee test suite to bind test model classes to various database instances when running the tests.

There are two sets of complementary methods:

- *Database.bind()* and *Model.bind()* - bind one or more models to a database.
- *Database.bind_ctx()* and *Model.bind_ctx()* - which are the same as their `bind()` counterparts, but return a context-manager and are useful when the database should only be changed temporarily.

As an example, we'll declare two models **without** specifying any database:

```
class User(Model):
    username = TextField()

class Tweet(Model):
    user = ForeignKeyField(User, backref='tweets')
    content = TextField()
    timestamp = TimestampField()
```

Bind the models to a database at run-time:

```
postgres_db = PostgresqlDatabase('my_app', user='postgres')
sqlite_db = SqliteDatabase('my_app.db')

# At this point, the User and Tweet models are NOT bound to any database.

# Let's bind them to the Postgres database:
postgres_db.bind([User, Tweet])

# Now we will temporarily bind them to the sqlite database:
with sqlite_db.bind_ctx([User, Tweet]):
    # User and Tweet are now bound to the sqlite database.
    assert User._meta.database is sqlite_db

# User and Tweet are once again bound to the Postgres database.
assert User._meta.database is postgres_db
```

The *Model.bind()* and *Model.bind_ctx()* methods work the same for binding a given model class:

```
# Bind the user model to the sqlite db. By default, Peewee will also
# bind any models that are related to User via foreign-key as well.
User.bind(sqlite_db)
```

(continues on next page)

(continued from previous page)

```
assert User._meta.database is sqlite_db
assert Tweet._meta.database is sqlite_db  # Related models bound too.

# Here we will temporarily bind *just* the User model to the postgres db.
with User.bind_ctx(postgres_db, bind_backrefs=False):
    assert User._meta.database is postgres_db
    assert Tweet._meta.database is sqlite_db  # Has not changed.

# And now User is back to being bound to the sqlite_db.
assert User._meta.database is sqlite_db
```

The *Testing Peewee Applications* section of this document also contains some examples of using the `bind()` methods.

1.6.10 Connection Management

To open a connection to a database, use the *`Database.connect()`* method:

```
>>> db = SqliteDatabase(':memory:')  # In-memory SQLite database.
>>> db.connect()
True
```

If we try to call `connect()` on an already-open database, we get a `OperationalError`:

```
>>> db.connect()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/charles/pypath/peewee.py", line 2390, in connect
    raise OperationalError('Connection already opened.')
peewee.OperationalError: Connection already opened.
```

To prevent this exception from being raised, we can call `connect()` with an additional argument, `reuse_if_open`:

```
>>> db.close()  # Close connection.
True
>>> db.connect()
True
>>> db.connect(reuse_if_open=True)
False
```

Note that the call to `connect()` returns `False` if the database connection was already open.

To close a connection, use the *`Database.close()`* method:

```
>>> db.close()
True
```

Calling `close()` on an already-closed connection will not result in an exception, but will return `False`:

```
>>> db.connect()  # Open connection.
True
>>> db.close()  # Close connection.
True
>>> db.close()  # Connection already closed, returns False.
False
```

You can test whether the database is closed using the *Database.is_closed()* method:

```
>>> db.is_closed()
True
```

Using autoconnect

It is not necessary to explicitly connect to the database before using it if the database is initialized with `autoconnect=True` (the default). Managing connections explicitly is considered a **best practice**, therefore you may consider disabling the `autoconnect` behavior.

It is very helpful to be explicit about your connection lifetimes. If the connection fails, for instance, the exception will be caught when the connection is being opened, rather than some arbitrary time later when a query is executed. Furthermore, if using a *connection pool*, it is necessary to call *connect()* and *close()* to ensure connections are recycled properly.

For the best guarantee of correctness, disable `autoconnect`:

```
db = PostgresqlDatabase('my_app', user='postgres', autoconnect=False)
```

Thread Safety

Peewee keeps track of the connection state using thread-local storage, making the Peewee *Database* object safe to use with multiple threads. Each thread will have it's own connection, and as a result any given thread will only have a single connection open at a given time.

Context managers

The database object itself can be used as a context-manager, which opens a connection for the duration of the wrapped block of code. Additionally, a transaction is opened at the start of the wrapped block and committed before the connection is closed (unless an error occurs, in which case the transaction is rolled back).

```
>>> db.is_closed()
True
>>> with db:
...     print(db.is_closed())  # db is open inside context manager.
...
False
>>> db.is_closed()  # db is closed.
True
```

If you want to manage transactions separately, you can use the *Database.connection_context()* context manager.

```
>>> with db.connection_context():
...     # db connection is open.
...     pass
...
>>> db.is_closed()  # db connection is closed.
True
```

The `connection_context()` method can also be used as a decorator:

```
@db.connection_context()
def prepare_database():
    # DB connection will be managed by the decorator, which opens
    # a connection, calls function, and closes upon returning.
    db.create_tables(MODELS)  # Create schema.
    load_fixture_data(db)
```

DB-API Connection Object

To obtain a reference to the underlying DB-API 2.0 connection, use the *Database.connection()* method. This method will return the currently-open connection object, if one exists, otherwise it will open a new connection.

```
>>> db.connection()
<sqlite3.Connection object at 0x7f94e9362f10>
```

1.6.11 Connection Pooling

Connection pooling is provided by the *pool module*, included in the *playhouse* extensions library. The pool supports:

- Timeout after which connections will be recycled.
- Upper bound on the number of open connections.

```
from playhouse.pool import PooledPostgresqlExtDatabase

db = PooledPostgresqlExtDatabase(
    'my_database',
    max_connections=8,
    stale_timeout=300,
    user='postgres')

class BaseModel(Model):
    class Meta:
        database = db
```

The following pooled database classes are available:

- *PooledPostgresqlDatabase*
- *PooledPostgresqlExtDatabase*
- *PooledMySQLDatabase*
- *PooledSqliteDatabase*
- *PooledSqliteExtDatabase*

For an in-depth discussion of peewee's connection pool, see the *Connection pool* section of the *playhouse* documentation.

1.6.12 Testing Peewee Applications

When writing tests for an application that uses Peewee, it may be desirable to use a special database for tests. Another common practice is to run tests against a clean database, which means ensuring tables are empty at the start of each test.

To bind your models to a database at run-time, you can use the following methods:

- *Database.bind_ctx()*, which returns a context-manager that will bind the given models to the database instance for the duration of the wrapped block.
- *Model.bind_ctx()*, which likewise returns a context-manager that binds the model (and optionally its dependencies) to the given database for the duration of the wrapped block.
- *Database.bind()*, which is a one-time operation that binds the models (and optionally its dependencies) to the given database.
- *Model.bind()*, which is a one-time operation that binds the model (and optionally its dependencies) to the given database.

Depending on your use-case, one of these options may make more sense. For the examples below, I will use *Model.bind()*.

Example test-case setup:

```
# tests.py
import unittest
from my_app.models import EventLog, Relationship, Tweet, User

MODELS = [User, Tweet, EventLog, Relationship]

# use an in-memory SQLite for tests.
test_db = SqliteDatabase(':memory:')

class BaseTestCase(unittest.TestCase):
    def setUp(self):
        # Bind model classes to test db. Since we have a complete list of
        # all models, we do not need to recursively bind dependencies.
        test_db.bind(MODELS, bind_refs=False, bind_backrefs=False)

        test_db.connect()
        test_db.create_tables(MODELS)

    def tearDown(self):
        # Not strictly necessary since SQLite in-memory databases only live
        # for the duration of the connection, and in the next step we close
        # the connection...but a good practice all the same.
        test_db.drop_tables(MODELS)

        # Close connection to db.
        test_db.close()

        # If we wanted, we could re-bind the models to their original
        # database here. But for tests this is probably not necessary.
```

As an aside, and speaking from experience, I recommend testing your application using the same database backend you use in production, so as to avoid any potential compatibility issues.

If you'd like to see some more examples of how to run tests using Peewee, check out Peewee's own test-suite.

1.6.13 Async with Gevent

gevent is recommended for doing asynchronous I/O with Postgresql or MySQL. Reasons I prefer gevent:

- No need for special-purpose "loop-aware" re-implementations of *everything*. Third-party libraries using asyncio usually have to re-implement layers and layers of code as well as re-implementing the protocols themselves.

- Gevent allows you to write your application in normal, clean, idiomatic Python. No need to litter every line with "async", "await" and other noise. No callbacks, futures, tasks, promises. No cruft.
- Gevent works with both Python 2 *and* Python 3.
- Gevent is *Pythonic*. Asyncio is an un-pythonic abomination.

Besides monkey-patching socket, no special steps are required if you are using **MySQL** with a pure Python driver like pymysql or are using mysql-connector in pure-python mode. MySQL drivers written in C will require special configuration which is beyond the scope of this document.

For **Postgres** and psycopg2, which is a C extension, you can use the following code snippet to register event hooks that will make your connection async:

```
from gevent.socket import wait_read, wait_write
from psycopg2 import extensions

# Call this function after monkey-patching socket (etc).
def patch_psycopg2():
    extensions.set_wait_callback(_psycopg2_gevent_callback)

def _psycopg2_gevent_callback(conn, timeout=None):
    while True:
        state = conn.poll()
        if state == extensions.POLL_OK:
            break
        elif state == extensions.POLL_READ:
            wait_read(conn.fileno(), timeout=timeout)
        elif state == extensions.POLL_WRITE:
            wait_write(conn.fileno(), timeout=timeout)
        else:
            raise ValueError('poll() returned unexpected result')
```

SQLite, because it is embedded in the Python application itself, does not do any socket operations that would be a candidate for non-blocking. Async has no effect one way or the other on SQLite databases.

1.6.14 Framework Integration

For web applications, it is common to open a connection when a request is received, and to close the connection when the response is delivered. In this section I will describe how to add hooks to your web app to ensure the database connection is handled properly.

These steps will ensure that regardless of whether you're using a simple SQLite database, or a pool of multiple Postgres connections, peewee will handle the connections correctly.

Note: Applications that receive lots of traffic may benefit from using a *connection pool* to mitigate the cost of setting up and tearing down connections on every request.

Flask

Flask and peewee are a great combo and my go-to for projects of any size. Flask provides two hooks which we will use to open and close our db connection. We'll open the connection when a request is received, then close it when the response is returned.

```
from flask import Flask
from peewee import *

database = SqliteDatabase('my_app.db')
app = Flask(__name__)

# This hook ensures that a connection is opened to handle any queries
# generated by the request.
@app.before_request
def _db_connect():
    database.connect()

# This hook ensures that the connection is closed when we've finished
# processing the request.
@app.teardown_request
def _db_close(exc):
    if not database.is_closed():
        database.close()
```

Django

While it's less common to see peewee used with Django, it is actually very easy to use the two. To manage your peewee database connections with Django, the easiest way in my opinion is to add a middleware to your app. The middleware should be the very first in the list of middlewares, to ensure it runs first when a request is handled, and last when the response is returned.

If you have a django project named *my_blog* and your peewee database is defined in the module `my_blog.db`, you might add the following middleware class:

```
# middleware.py
from my_blog.db import database  # Import the peewee database instance.


def PeeweeConnectionMiddleware(get_response):
    def middleware(request):
        database.connect()
        try:
            response = get_response(request)
        finally:
            if not database.is_closed():
                database.close()
        return response
    return middleware


# Older Django < 1.10 middleware.
class PeeweeConnectionMiddleware(object):
    def process_request(self, request):
        database.connect()

    def process_response(self, request, response):
        if not database.is_closed():
            database.close()
        return response
```

To ensure this middleware gets executed, add it to your `settings` module:

```
# settings.py
MIDDLEWARE_CLASSES = (
    # Our custom middleware appears first in the list.
    'my_blog.middleware.PeeweeConnectionMiddleware',

    # These are the default Django 1.7 middlewares. Yours may differ,
    # but the important this is that our Peewee middleware comes first.
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
)

# ... other Django settings ...
```

Bottle

I haven't used bottle myself, but looking at the documentation I believe the following code should ensure the database connections are properly managed:

```
# app.py
from bottle import hook  #, route, etc, etc.
from peewee import *

db = SqliteDatabase('my-bottle-app.db')

@hook('before_request')
def _connect_db():
    db.connect()

@hook('after_request')
def _close_db():
    if not db.is_closed():
        db.close()

# Rest of your bottle app goes here.
```

Web.py

See the documentation for application processors.

```
db = SqliteDatabase('my_webpy_app.db')

def connection_processor(handler):
    db.connect()
    try:
        return handler()
    finally:
        if not db.is_closed():
            db.close()

app.add_processor(connection_processor)
```

Tornado

It looks like Tornado's `RequestHandler` class implements two hooks which can be used to open and close connections when a request is handled.

```
from tornado.web import RequestHandler

db = SqliteDatabase('my_db.db')

class PeeweeRequestHandler(RequestHandler):
    def prepare(self):
        db.connect()
        return super(PeeweeRequestHandler, self).prepare()

    def on_finish(self):
        if not db.is_closed():
            db.close()
        return super(PeeweeRequestHandler, self).on_finish()
```

In your app, instead of extending the default `RequestHandler`, now you can extend `PeeweeRequestHandler`.

Note that this does not address how to use peewee asynchronously with Tornado or another event loop.

Wheezy.web

The connection handling code can be placed in a middleware.

```
def peewee_middleware(request, following):
    db.connect()
    try:
        response = following(request)
    finally:
        if not db.is_closed():
            db.close()
    return response

app = WSGIApplication(middleware=[
    lambda x: peewee_middleware,
    # ... other middlewares ...
])
```

Thanks to GitHub user *@tuukkamustonen* for submitting this code.

Falcon

The connection handling code can be placed in a middleware component.

```
import falcon
from peewee import *

database = SqliteDatabase('my_app.db')

class PeeweeConnectionMiddleware(object):
    def process_request(self, req, resp):
        database.connect()
```

(continues on next page)

(continued from previous page)

```
    def process_response(self, req, resp, resource):
        if not database.is_closed():
            database.close()

application = falcon.API(middleware=[
    PeeweeConnectionMiddleware(),
    # ... other middlewares ...
])
```

Pyramid

Set up a Request factory that handles database connection lifetime as follows:

```
from pyramid.request import Request

db = SqliteDatabase('pyramidapp.db')

class MyRequest(Request):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        db.connect()
        self.add_finished_callback(self.finish)

    def finish(self, request):
        if not db.is_closed():
            db.close()
```

In your application *main()* make sure *MyRequest* is used as *request_factory*:

```
def main(global_settings, **settings):
    config = Configurator(settings=settings, ...)
    config.set_request_factory(MyRequest)
```

CherryPy

See Publish/Subscribe pattern.

```
def _db_connect():
    db.connect()

def _db_close():
    if not db.is_closed():
        db.close()

cherrypy.engine.subscribe('before_request', _db_connect)
cherrypy.engine.subscribe('after_request', _db_close)
```

Sanic

In Sanic, the connection handling code can be placed in the request and response middleware sanic middleware.

```
# app.py
@app.middleware('request')
async def handle_request(request):
    db.connect()

@app.middleware('response')
async def handle_response(request, response):
    if not db.is_closed():
        db.close()
```

FastAPI

Similar to Flask, FastAPI provides two event based hooks which we will use to open and close our db connection. We'll open the connection when a request is received, then close it when the response is returned.

```
from fastapi import FastAPI
from peewee import *

db = SqliteDatabase('my_app.db')
app = FastAPI()

# This hook ensures that a connection is opened to handle any queries
# generated by the request.
@app.on_event("startup")
def startup():
    db.connect()


# This hook ensures that the connection is closed when we've finished
# processing the request.
@app.on_event("shutdown")
def shutdown():
    if not db.is_closed():
        db.close()
```

Other frameworks

Don't see your framework here? Please open a GitHub ticket and I'll see about adding a section, or better yet, submit a documentation pull-request.

1.6.15 Executing Queries

SQL queries will typically be executed by calling `execute()` on a query constructed using the query-builder APIs (or by simply iterating over a query object in the case of a *`Select`* query). For cases where you wish to execute SQL directly, you can use the *`Database.execute_sql()`* method.

```
db = SqliteDatabase('my_app.db')
db.connect()

# Example of executing a simple query and ignoring the results.
db.execute_sql("ATTACH DATABASE ':memory:' AS cache;")

# Example of iterating over the results of a query using the cursor.
```

(continues on next page)

(continued from previous page)

```
cursor = db.execute_sql('SELECT * FROM users WHERE status = ?', (ACTIVE,))
for row in cursor.fetchall():
    # Do something with row, which is a tuple containing column data.
    pass
```

1.6.16 Managing Transactions

Peewee provides several interfaces for working with transactions. The most general is the *Database.atomic()* method, which also supports nested transactions. *atomic()* blocks will be run in a transaction or savepoint, depending on the level of nesting.

If an exception occurs in a wrapped block, the current transaction/savepoint will be rolled back. Otherwise the statements will be committed at the end of the wrapped block.

Note: While inside a block wrapped by the *atomic()* context manager, you can explicitly rollback or commit at any point by calling `Transaction.rollback()` or `Transaction.commit()`. When you do this inside a wrapped block of code, a new transaction will be started automatically.

```
with db.atomic() as transaction:  # Opens new transaction.
    try:
        save_some_objects()
    except ErrorSavingData:
        # Because this block of code is wrapped with "atomic", a
        # new transaction will begin automatically after the call
        # to rollback().
        transaction.rollback()
        error_saving = True

    create_report(error_saving=error_saving)
    # Note: no need to call commit. Since this marks the end of the
    # wrapped block of code, the `atomic` context manager will
    # automatically call commit for us.
```

Note: *atomic()* can be used as either a **context manager** or a **decorator**.

Context manager

Using `atomic` as context manager:

```
db = SqliteDatabase(':memory:')

with db.atomic() as txn:
    # This is the outer-most level, so this block corresponds to
    # a transaction.
    User.create(username='charlie')

    with db.atomic() as nested_txn:
        # This block corresponds to a savepoint.
        User.create(username='huey')
```

(continues on next page)

(continued from previous page)

```
        # This will roll back the above create() query.
        nested_txn.rollback()

    User.create(username='mickey')

# When the block ends, the transaction is committed (assuming no error
# occurs). At that point there will be two users, "charlie" and "mickey".
```

You can use the `atomic` method to perform *get or create* operations as well:

```
try:
    with db.atomic():
        user = User.create(username=username)
    return 'Success'
except peewee.IntegrityError:
    return 'Failure: %s is already in use.' % username
```

Decorator

Using `atomic` as a decorator:

```
@db.atomic()
def create_user(username):
    # This statement will run in a transaction. If the caller is already
    # running in an `atomic` block, then a savepoint will be used instead.
    return User.create(username=username)

create_user('charlie')
```

Nesting Transactions

atomic() provides transparent nesting of transactions. When using *atomic()*, the outer-most call will be wrapped in a transaction, and any nested calls will use savepoints.

```
with db.atomic() as txn:
    perform_operation()

    with db.atomic() as nested_txn:
        perform_another_operation()
```

Peewee supports nested transactions through the use of savepoints (for more information, see *savepoint()*).

Explicit transaction

If you wish to explicitly run code in a transaction, you can use *transaction()*. Like *atomic()*, *transaction()* can be used as a context manager or as a decorator.

If an exception occurs in a wrapped block, the transaction will be rolled back. Otherwise the statements will be committed at the end of the wrapped block.

```
db = SqliteDatabase(':memory:')
```

(continues on next page)

(continued from previous page)

```
with db.transaction() as txn:
    # Delete the user and their associated tweets.
    user.delete_instance(recursive=True)
```

Transactions can be explicitly committed or rolled-back within the wrapped block. When this happens, a new transaction will be started.

```
with db.transaction() as txn:
    User.create(username='mickey')
    txn.commit()  # Changes are saved and a new transaction begins.
    User.create(username='huey')

    # Roll back. "huey" will not be saved, but since "mickey" was already
    # committed, that row will remain in the database.
    txn.rollback()

with db.transaction() as txn:
    User.create(username='whiskers')
    # Roll back changes, which removes "whiskers".
    txn.rollback()

    # Create a new row for "mr. whiskers" which will be implicitly committed
    # at the end of the `with` block.
    User.create(username='mr. whiskers')
```

Note: If you attempt to nest transactions with peewee using the *transaction()* context manager, only the outer-most transaction will be used. However if an exception occurs in a nested block, this can lead to unpredictable behavior, so it is strongly recommended that you use *atomic()*.

Explicit Savepoints

Just as you can explicitly create transactions, you can also explicitly create savepoints using the *savepoint()* method. Savepoints must occur within a transaction, but can be nested arbitrarily deep.

```
with db.transaction() as txn:
    with db.savepoint() as sp:
        User.create(username='mickey')

    with db.savepoint() as sp2:
        User.create(username='zaizee')
        sp2.rollback()  # "zaizee" will not be saved, but "mickey" will be.
```

Warning: If you manually commit or roll back a savepoint, a new savepoint **will not** automatically be created. This differs from the behavior of `transaction`, which will automatically open a new transaction after manual commit/rollback.

Autocommit Mode

By default, Peewee operates in *autocommit mode*, such that any statements executed outside of a transaction are run in their own transaction. To group multiple statements into a transaction, Peewee provides the *atomic()* context-

manager/decorator. This should cover all use-cases, but in the unlikely event you want to temporarily disable Peewee's transaction management completely, you can use the *Database.manual_commit()* context-manager/decorator.

Here is how you might emulate the behavior of the *transaction()* context manager:

```
with db.manual_commit():
    db.begin()  # Have to begin transaction explicitly.
    try:
        user.delete_instance(recursive=True)
    except:
        db.rollback()  # Rollback! An error occurred.
        raise
    else:
        try:
            db.commit()  # Commit changes.
        except:
            db.rollback()
            raise
```

Again – I don't anticipate anyone needing this, but it's here just in case.

1.6.17 Database Errors

The Python DB-API 2.0 spec describes several types of exceptions. Because most database drivers have their own implementations of these exceptions, Peewee simplifies things by providing its own wrappers around any implementation-specific exception classes. That way, you don't need to worry about importing any special exception classes, you can just use the ones from peewee:

- `DatabaseError`
- `DataError`
- `IntegrityError`
- `InterfaceError`
- `InternalError`
- `NotSupportedError`
- `OperationalError`
- `ProgrammingError`

Note: All of these error classes extend `PeeweeException`.

1.6.18 Logging queries

All queries are logged to the *peewee* namespace using the standard library `logging` module. Queries are logged using the *DEBUG* level. If you're interested in doing something with the queries, you can simply register a handler.

```
# Print all queries to stderr.
import logging
logger = logging.getLogger('peewee')
logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)
```

1.6.19 Adding a new Database Driver

Peewee comes with built-in support for Postgres, MySQL and SQLite. These databases are very popular and run the gamut from fast, embeddable databases to heavyweight servers suitable for large-scale deployments. That being said, there are a ton of cool databases out there and adding support for your database-of-choice should be really easy, provided the driver supports the DB-API 2.0 spec.

The DB-API 2.0 spec should be familiar to you if you've used the standard library sqlite3 driver, psycopg2 or the like. Peewee currently relies on a handful of parts:

- *Connection.commit*
- *Connection.execute*
- *Connection.rollback*
- *Cursor.description*
- *Cursor.fetchone*

These methods are generally wrapped up in higher-level abstractions and exposed by the *Database*, so even if your driver doesn't do these exactly you can still get a lot of mileage out of peewee. An example is the apsw sqlite driver in the "playhouse" module.

The first thing is to provide a subclass of *Database* that will open a connection.

```
from peewee import Database
import foodb  # Our fictional DB-API 2.0 driver.


class FooDatabase(Database):
    def _connect(self, database, **kwargs):
        return foodb.connect(database, **kwargs)
```

The *Database* provides a higher-level API and is responsible for executing queries, creating tables and indexes, and introspecting the database to get lists of tables. The above implementation is the absolute minimum needed, though some features will not work – for best results you will want to additionally add a method for extracting a list of tables and indexes for a table from the database. We'll pretend that `FooDB` is a lot like MySQL and has special "SHOW" statements:

```
class FooDatabase(Database):
    def _connect(self, database, **kwargs):
        return foodb.connect(database, **kwargs)

    def get_tables(self):
        res = self.execute('SHOW TABLES;')
        return [r[0] for r in res.fetchall()]
```

Other things the database handles that are not covered here include:

- *last_insert_id()* and *rows_affected()*
- `param` and `quote`, which tell the SQL-generating code how to add parameter placeholders and quote entity names.
- `field_types` for mapping data-types like INT or TEXT to their vendor-specific type names.
- `operations` for mapping operations such as "LIKE/ILIKE" to their database equivalent

Refer to the *Database* API reference or the source code. for details.

Note: If your driver conforms to the DB-API 2.0 spec, there shouldn't be much work needed to get up and running.

Our new database can be used just like any of the other database subclasses:

```
from peewee import *
from foodb_ext import FooDatabase

db = FooDatabase('my_database', user='foo', password='secret')

class BaseModel(Model):
    class Meta:
        database = db

class Blog(BaseModel):
    title = CharField()
    contents = TextField()
    pub_date = DateTimeField()
```

1.7 Models and Fields

Model classes, *Field* instances and model instances all map to database concepts:

Thing	Corresponds to...
Model class	Database table
Field instance	Column on a table
Model instance	Row in a database table

The following code shows the typical way you will define your database connection and model classes.

```
import datetime
from peewee import *

db = SqliteDatabase('my_app.db')

class BaseModel(Model):
    class Meta:
        database = db

class User(BaseModel):
    username = CharField(unique=True)

class Tweet(BaseModel):
    user = ForeignKeyField(User, backref='tweets')
    message = TextField()
    created_date = DateTimeField(default=datetime.datetime.now)
    is_published = BooleanField(default=True)
```

1. Create an instance of a *Database*.

   ```
   db = SqliteDatabase('my_app.db')
   ```

   The `db` object will be used to manage the connections to the Sqlite database. In this example we're using *SqliteDatabase*, but you could also use one of the other *database engines*.

2. Create a base model class which specifies our database.

```
class BaseModel(Model):
    class Meta:
        database = db
```

   It is good practice to define a base model class which establishes the database connection. This makes your code DRY as you will not have to specify the database for subsequent models.

   Model configuration is kept namespaced in a special class called `Meta`. This convention is borrowed from Django. *Meta* configuration is passed on to subclasses, so our project's models will all subclass *BaseModel*. There are *many different attributes* you can configure using *Model.Meta*.

3. Define a model class.

```
class User(BaseModel):
    username = CharField(unique=True)
```

   Model definition uses the declarative style seen in other popular ORMs like SQLAlchemy or Django. Note that we are extending the *BaseModel* class so the *User* model will inherit the database connection.

   We have explicitly defined a single *username* column with a unique constraint. Because we have not specified a primary key, peewee will automatically add an auto-incrementing integer primary key field named *id*.

Note: If you would like to start using peewee with an existing database, you can use *pwiz, a model generator* to automatically generate model definitions.

1.7.1 Fields

The *Field* class is used to describe the mapping of *Model* attributes to database columns. Each field type has a corresponding SQL storage class (i.e. varchar, int), and conversion between python data types and underlying storage is handled transparently.

When creating a *Model* class, fields are defined as class attributes. This should look familiar to users of the django framework. Here's an example:

```
class User(Model):
    username = CharField()
    join_date = DateTimeField()
    about_me = TextField()
```

In the above example, because none of the fields are initialized with `primary_key=True`, an auto-incrementing primary key will automatically be created and named "id". Peewee uses *AutoField* to signify an auto-incrementing integer primary key, which implies `primary_key=True`.

There is one special type of field, *ForeignKeyField*, which allows you to represent foreign-key relationships between models in an intuitive way:

```
class Message(Model):
    user = ForeignKeyField(User, backref='messages')
    body = TextField()
    send_date = DateTimeField(default=datetime.datetime.now)
```

This allows you to write code like the following:

```
>>> print(some_message.user.username)
Some User

>>> for message in some_user.messages:
...     print(message.body)
some message
another message
yet another message
```

Note: Refer to the *Relationships and Joins* document for an in-depth discussion of foreign-keys, joins and relationships between models.

For full documentation on fields, see the *Fields API notes*

Field types table

Field Type	Sqlite	Postgresql	MySQL
`AutoField`	integer	serial	integer
`BigAutoField`	integer	bigserial	bigint
`IntegerField`	integer	integer	integer
`BigIntegerField`	integer	bigint	bigint
`SmallIntegerField`	integer	smallint	smallint
`IdentityField`	not supported	int identity	not supported
`FloatField`	real	real	real
`DoubleField`	real	double precision	double precision
`DecimalField`	decimal	numeric	numeric
`CharField`	varchar	varchar	varchar
`FixedCharField`	char	char	char
`TextField`	text	text	text
`BlobField`	blob	bytea	blob
`BitField`	integer	bigint	bigint
`BigBitField`	blob	bytea	blob
`UUIDField`	text	uuid	varchar(40)
`BinaryUUIDField`	blob	bytea	varbinary(16)
`DateTimeField`	datetime	timestamp	datetime
`DateField`	date	date	date
`TimeField`	time	time	time
`TimestampField`	integer	integer	integer
`IPField`	integer	bigint	bigint
`BooleanField`	integer	boolean	bool
`BareField`	untyped	not supported	not supported
`ForeignKeyField`	integer	integer	integer

Note: Don't see the field you're looking for in the above table? It's easy to create custom field types and use them with your models.

- *Creating a custom field*
- *Database*, particularly the `fields` parameter.

Field initialization arguments

Parameters accepted by all field types and their default values:

- `null = False` – allow null values
- `index = False` – create an index on this column
- `unique = False` – create a unique index on this column. See also *adding composite indexes*.
- `column_name = None` – explicitly specify the column name in the database.
- `default = None` – any value or callable to use as a default for uninitialized models
- `primary_key = False` – primary key for the table
- `constraints = None` - one or more constraints, e.g. `[Check('price > 0')]`
- `sequence = None` – sequence name (if backend supports it)
- `collation = None` – collation to use for ordering the field / index
- `unindexed = False` – indicate field on virtual table should be unindexed (**SQLite-only**)
- `choices = None` – optional iterable containing 2-tuples of `value`, `display`
- `help_text = None` – string representing any helpful text for this field
- `verbose_name = None` – string representing the "user-friendly" name of this field
- `index_type = None` – specify a custom index-type, e.g. for Postgres you might specify a `'BRIN'` or `'GIN'` index.

Some fields take special parameters...

Field type	Special Parameters
CharField	`max_length`
FixedCharField	`max_length`
DateTimeField	`formats`
DateField	`formats`
TimeField	`formats`
TimestampField	`resolution`, `utc`
DecimalField	`max_digits`, `decimal_places`, `auto_round`, `rounding`
ForeignKeyField	`model`, `field`, `backref`, `on_delete`, `on_update`, `deferrable` `lazy_load`
BareField	`adapt`

Note: Both `default` and `choices` could be implemented at the database level as *DEFAULT* and *CHECK CONSTRAINT* respectively, but any application change would require a schema change. Because of this, `default` is implemented purely in python and `choices` are not validated but exist for metadata purposes only.

To add database (server-side) constraints, use the `constraints` parameter.

Default field values

Peewee can provide default values for fields when objects are created. For example to have an `IntegerField` default to zero rather than `NULL`, you could declare the field with a default value:

```
class Message(Model):
    context = TextField()
    read_count = IntegerField(default=0)
```

In some instances it may make sense for the default value to be dynamic. A common scenario is using the current date and time. Peewee allows you to specify a function in these cases, whose return value will be used when the object is created. Note we only provide the function, we do not actually *call* it:

```
class Message(Model):
    context = TextField()
    timestamp = DateTimeField(default=datetime.datetime.now)
```

Note: If you are using a field that accepts a mutable type (*list*, *dict*, etc), and would like to provide a default, it is a good idea to wrap your default value in a simple function so that multiple model instances are not sharing a reference to the same underlying object:

```
def house_defaults():
    return {'beds': 0, 'baths': 0}

class House(Model):
    number = TextField()
    street = TextField()
    attributes = JSONField(default=house_defaults)
```

The database can also provide the default value for a field. While peewee does not explicitly provide an API for setting a server-side default value, you can use the `constraints` parameter to specify the server default:

```
class Message(Model):
    context = TextField()
    timestamp = DateTimeField(constraints=[SQL('DEFAULT CURRENT_TIMESTAMP')])
```

Note: **Remember:** when using the `default` parameter, the values are set by Peewee rather than being a part of the actual table and column definition.

ForeignKeyField

`ForeignKeyField` is a special field type that allows one model to reference another. Typically a foreign key will contain the primary key of the model it relates to (but you can specify a particular column by specifying a `field`).

Foreign keys allow data to be normalized. In our example models, there is a foreign key from `Tweet` to `User`. This means that all the users are stored in their own table, as are the tweets, and the foreign key from tweet to user allows each tweet to *point* to a particular user object.

Note: Refer to the *Relationships and Joins* document for an in-depth discussion of foreign keys, joins and relationships between models.

In peewee, accessing the value of a *`ForeignKeyField`* will return the entire related object, e.g.:

```
tweets = (Tweet
          .select(Tweet, User)
```

(continues on next page)

(continued from previous page)

```
            .join(User)
            .order_by(Tweet.created_date.desc()))
for tweet in tweets:
    print(tweet.user.username, tweet.message)
```

Note: In the example above the `User` data was selected as part of the query. For more examples of this technique, see the *Avoiding N+1* document.

If we did not select the `User`, though, then an **additional query** would be issued to fetch the associated `User` data:

```
tweets = Tweet.select().order_by(Tweet.created_date.desc())
for tweet in tweets:
    # WARNING: an additional query will be issued for EACH tweet
    # to fetch the associated User data.
    print(tweet.user.username, tweet.message)
```

Sometimes you only need the associated primary key value from the foreign key column. In this case, Peewee follows the convention established by Django, of allowing you to access the raw foreign key value by appending `"_id"` to the foreign key field's name:

```
tweets = Tweet.select()
for tweet in tweets:
    # Instead of "tweet.user", we will just get the raw ID value stored
    # in the column.
    print(tweet.user_id, tweet.message)
```

To prevent accidentally resolving a foreign-key and triggering an additional query, *ForeignKeyField* supports an initialization paramater `lazy_load` which, when disabled, behaves like the `"_id"` attribute. For example:

```
class Tweet(Model):
    # ... same fields, except we declare the user FK to have
    # lazy-load disabled:
    user = ForeignKeyField(User, backref='tweets', lazy_load=False)

for tweet in Tweet.select():
    print(tweet.user, tweet.message)

# With lazy-load disabled, accessing tweet.user will not perform an extra
# query and the user ID value is returned instead.
# e.g.:
# 1  tweet from user1
# 1  another from user1
# 2  tweet from user2

# However, if we eagerly load the related user object, then the user
# foreign key will behave like usual:
for tweet in Tweet.select(Tweet, User).join(User):
    print(tweet.user.username, tweet.message)

# user1  tweet from user1
# user1  another from user1
# user2  tweet from user1
```

ForeignKeyField Back-references

ForeignKeyField allows for a backreferencing property to be bound to the target model. Implicitly, this property will be named `classname_set`, where `classname` is the lowercase name of the class, but can be overridden using the parameter `backref`:

```
class Message(Model):
    from_user = ForeignKeyField(User, backref='outbox')
    to_user = ForeignKeyField(User, backref='inbox')
    text = TextField()

for message in some_user.outbox:
    # We are iterating over all Messages whose from_user is some_user.
    print(message)

for message in some_user.inbox:
    # We are iterating over all Messages whose to_user is some_user
    print(message)
```

DateTimeField, DateField and TimeField

The three fields devoted to working with dates and times have special properties which allow access to things like the year, month, hour, etc.

DateField has properties for:

- `year`
- `month`
- `day`

TimeField has properties for:

- `hour`
- `minute`
- `second`

DateTimeField has all of the above.

These properties can be used just like any other expression. Let's say we have an events calendar and want to highlight all the days in the current month that have an event attached:

```
# Get the current time.
now = datetime.datetime.now()

# Get days that have events for the current month.
Event.select(Event.event_date.day.alias('day')).where(
    (Event.event_date.year == now.year) &
    (Event.event_date.month == now.month))
```

Note: SQLite does not have a native date type, so dates are stored in formatted text columns. To ensure that comparisons work correctly, the dates need to be formatted so they are sorted lexicographically. That is why they are stored, by default, as `YYYY-MM-DD HH:MM:SS`.

BitField and BigBitField

The *BitField* and *BigBitField* are new as of 3.0.0. The former provides a subclass of *IntegerField* that is suitable for storing feature toggles as an integer bitmask. The latter is suitable for storing a bitmap for a large data-set, e.g. expressing membership or bitmap-type data.

As an example of using *BitField*, let's say we have a *Post* model and we wish to store certain True/False flags about how the post. We could store all these feature toggles in their own *BooleanField* objects, or we could use *BitField* instead:

```
class Post(Model):
    content = TextField()
    flags = BitField()

    is_favorite = flags.flag(1)
    is_sticky = flags.flag(2)
    is_minimized = flags.flag(4)
    is_deleted = flags.flag(8)
```

Using these flags is quite simple:

```
>>> p = Post()
>>> p.is_sticky = True
>>> p.is_minimized = True
>>> print(p.flags)  # Prints 4 | 2 --> "6"
6
>>> p.is_favorite
False
>>> p.is_sticky
True
```

We can also use the flags on the Post class to build expressions in queries:

```
# Generates a WHERE clause that looks like:
# WHERE (post.flags & 1 != 0)
favorites = Post.select().where(Post.is_favorite)

# Query for sticky + favorite posts:
sticky_faves = Post.select().where(Post.is_sticky & Post.is_favorite)
```

Since the *BitField* is stored in an integer, there is a maximum of 64 flags you can represent (64-bits is common size of integer column). For storing arbitrarily large bitmaps, you can instead use *BigBitField*, which uses an automatically managed buffer of bytes, stored in a *BlobField*.

Example usage:

```
class Bitmap(Model):
    data = BigBitField()

bitmap = Bitmap()

# Sets the ith bit, e.g. the 1st bit, the 11th bit, the 63rd, etc.
bits_to_set = (1, 11, 63, 31, 55, 48, 100, 99)
for bit_idx in bits_to_set:
    bitmap.data.set_bit(bit_idx)

# We can test whether a bit is set using "is_set":
assert bitmap.data.is_set(11)
```

(continues on next page)

(continued from previous page)

```
assert not bitmap.data.is_set(12)

# We can clear a bit:
bitmap.data.clear_bit(11)
assert not bitmap.data.is_set(11)

# We can also "toggle" a bit. Recall that the 63rd bit was set earlier.
assert bitmap.data.toggle_bit(63) is False
assert bitmap.data.toggle_bit(63) is True
assert bitmap.data.is_set(63)
```

BareField

The *BareField* class is intended to be used only with SQLite. Since SQLite uses dynamic typing and data-types are not enforced, it can be perfectly fine to declare fields without *any* data-type. In those cases you can use *BareField*. It is also common for SQLite virtual tables to use meta-columns or untyped columns, so for those cases as well you may wish to use an untyped field (although for full-text search, you should use *SearchField* instead!).

BareField accepts a special parameter `adapt`. This parameter is a function that takes a value coming from the database and converts it into the appropriate Python type. For instance, if you have a virtual table with an un-typed column but you know that it will return `int` objects, you can specify `adapt=int`.

Example:

```
db = SqliteDatabase(':memory:')

class Junk(Model):
    anything = BareField()

    class Meta:
        database = db

# Store multiple data-types in the Junk.anything column:
Junk.create(anything='a string')
Junk.create(anything=12345)
Junk.create(anything=3.14159)
```

Creating a custom field

It is easy to add support for custom field types in peewee. In this example we will create a UUID field for postgresql (which has a native UUID column type).

To add a custom field type you need to first identify what type of column the field data will be stored in. If you just want to add python behavior atop, say, a decimal field (for instance to make a currency field) you would just subclass *DecimalField*. On the other hand, if the database offers a custom column type you will need to let peewee know. This is controlled by the `Field.field_type` attribute.

Note: Peewee ships with a *UUIDField*, the following code is intended only as an example.

Let's start by defining our UUID field:

```
class UUIDField(Field):
    field_type = 'uuid'
```

We will store the UUIDs in a native UUID column. Since psycopg2 treats the data as a string by default, we will add two methods to the field to handle:

- The data coming out of the database to be used in our application
- The data from our python app going into the database

```
import uuid

class UUIDField(Field):
    field_type = 'uuid'

    def db_value(self, value):
        return value.hex  # convert UUID to hex string.

    def python_value(self, value):
        return uuid.UUID(value) # convert hex string to UUID
```

This step is optional. By default, the `field_type` value will be used for the columns data-type in the database schema. If you need to support multiple databases which use different data-types for your field-data, we need to let the database know how to map this *uuid* label to an actual *uuid* column type in the database. Specify the overrides in the *Database* constructor:

```
# Postgres, we use UUID data-type.
db = PostgresqlDatabase('my_db', field_types={'uuid': 'uuid'})

# Sqlite doesn't have a UUID type, so we use text type.
db = SqliteDatabase('my_db', field_types={'uuid': 'text'})
```

That is it! Some fields may support exotic operations, like the postgresql HStore field acts like a key/value store and has custom operators for things like *contains* and *update*. You can specify *custom operations* as well. For example code, check out the source code for the *HStoreField*, in `playhouse.postgres_ext`.

Field-naming conflicts

Model classes implement a number of class- and instance-methods, for example *Model.save()* or *Model.create()*. If you declare a field whose name coincides with a model method, it could cause problems. Consider:

```
class LogEntry(Model):
    event = TextField()
    create = TimestampField()  # Uh-oh.
    update = TimestampField()  # Uh-oh.
```

To avoid this problem while still using the desired column name in the database schema, explicitly specify the `column_name` while providing an alternative name for the field attribute:

```
class LogEntry(Model):
    event = TextField()
    create_ = TimestampField(column_name='create')
    update_ = TimestampField(column_name='update')
```

1.7.2 Creating model tables

In order to start using our models, its necessary to open a connection to the database and create the tables first. Peewee will run the necessary *CREATE TABLE* queries, additionally creating any constraints and indexes.

```
# Connect to our database.
db.connect()

# Create the tables.
db.create_tables([User, Tweet])
```

Note: Strictly speaking, it is not necessary to call `connect()` but it is good practice to be explicit. That way if something goes wrong, the error occurs at the connect step, rather than some arbitrary time later.

Note: By default, Peewee includes an `IF NOT EXISTS` clause when creating tables. If you want to disable this, specify `safe=False`.

After you have created your tables, if you choose to modify your database schema (by adding, removing or otherwise changing the columns) you will need to either:

- Drop the table and re-create it.
- Run one or more *ALTER TABLE* queries. Peewee comes with a schema migration tool which can greatly simplify this. Check the *schema migrations* docs for details.

1.7.3 Model options and table metadata

In order not to pollute the model namespace, model-specific configuration is placed in a special class called *Meta* (a convention borrowed from the django framework):

```
from peewee import *

contacts_db = SqliteDatabase('contacts.db')

class Person(Model):
    name = CharField()

    class Meta:
        database = contacts_db
```

This instructs peewee that whenever a query is executed on *Person* to use the contacts database.

Note: Take a look at *the sample models* - you will notice that we created a `BaseModel` that defined the database, and then extended. This is the preferred way to define a database and create models.

Once the class is defined, you should not access `ModelClass.Meta`, but instead use `ModelClass._meta`:

```
>>> Person.Meta
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Person' has no attribute 'Meta'

>>> Person._meta
<peewee.ModelOptions object at 0x7f51a2f03790>
```

The `ModelOptions` class implements several methods which may be of use for retrieving model metadata (such as lists of fields, foreign key relationships, and more).

```
>>> Person._meta.fields
{'id': <peewee.AutoField object at 0x7f51a2e92750>,
 'name': <peewee.CharField object at 0x7f51a2f0a510>}

>>> Person._meta.primary_key
<peewee.AutoField object at 0x7f51a2e92750>

>>> Person._meta.database
<peewee.SqliteDatabase object at 0x7f519bff6dd0>
```

There are several options you can specify as `Meta` attributes. While most options are inheritable, some are table-specific and will not be inherited by subclasses.

Option	Meaning	Inheritable?
`database`	database for model	yes
`table_name`	name of the table to store data	no
`table_function`	function to generate table name dynamically	yes
`indexes`	a list of fields to index	yes
`primary_key`	a *`CompositeKey`* instance	yes
`constraints`	a list of table constraints	yes
`schema`	the database schema for the model	yes
`only_save_dirty`	when calling model.save(), only save dirty fields	yes
`options`	dictionary of options for create table extensions	yes
`table_settings`	list of setting strings to go after close parentheses	yes
`temporary`	indicate temporary table	yes
`legacy_table_names`	use legacy table name generation (enabled by default)	yes
`depends_on`	indicate this table depends on another for creation	no
`without_rowid`	indicate table should not have rowid (SQLite only)	no

Here is an example showing inheritable versus non-inheritable attributes:

```
>>> db = SqliteDatabase(':memory:')
>>> class ModelOne(Model):
...     class Meta:
...         database = db
...         table_name = 'model_one_tbl'
...
>>> class ModelTwo(ModelOne):
...     pass
...
>>> ModelOne._meta.database is ModelTwo._meta.database
True
>>> ModelOne._meta.table_name == ModelTwo._meta.table_name
False
```

Meta.primary_key

The `Meta.primary_key` attribute is used to specify either a *`CompositeKey`* or to indicate that the model has *no* primary key. Composite primary keys are discussed in more detail here: *Composite primary keys*.

To indicate that a model should not have a primary key, then set `primary_key = False`.

Examples:

```
class BlogToTag(Model):
    """A simple "through" table for many-to-many relationship."""
    blog = ForeignKeyField(Blog)
    tag = ForeignKeyField(Tag)

    class Meta:
        primary_key = CompositeKey('blog', 'tag')

class NoPrimaryKey(Model):
    data = IntegerField()

    class Meta:
        primary_key = False
```

Table Names

By default Peewee will automatically generate a table name based on the name of your model class. The way the table-name is generated depends on the value of `Meta.legacy_table_names`. By default, `legacy_table_names=True` so as to avoid breaking backwards-compatibility. However, if you wish to use the new and improved table-name generation, you can specify `legacy_table_names=False`.

This table shows the differences in how a model name is converted to a SQL table name, depending on the value of `legacy_table_names`:

Model name	legacy_table_names=True	legacy_table_names=False (new)
User	user	user
UserProfile	userprofile	user_profile
APIResponse	apiresponse	api_response
WebHTTPRequest	webhttprequest	web_http_request
mixedCamelCase	mixedcamelcase	mixed_camel_case
Name2Numbers3XYZ	name2numbers3xyz	name2_numbers3_xyz

Attention: To preserve backwards-compatibility, the current release (Peewee 3.x) specifies `legacy_table_names=True` by default.

In the next major release (Peewee 4.0), `legacy_table_names` will have a default value of `False`.

To explicitly specify the table name for a model class, use the `table_name` Meta option. This feature can be useful for dealing with pre-existing database schemas that may have used awkward naming conventions:

```
class UserProfile(Model):
    class Meta:
        table_name = 'user_profile_tbl'
```

If you wish to implement your own naming convention, you can specify the `table_function` Meta option. This function will be called with your model class and should return the desired table name as a string. Suppose our company specifies that table names should be lower-cased and end with "_tbl", we can implement this as a table function:

```
def make_table_name(model_class):
    model_name = model_class.__name__
    return model_name.lower() + '_tbl'
```

(continues on next page)

(continued from previous page)

```
class BaseModel(Model):
    class Meta:
        table_function = make_table_name

class User(BaseModel):
    # table_name will be "user_tbl".

class UserProfile(BaseModel):
    # table_name will be "userprofile_tbl".
```

1.7.4 Indexes and Constraints

Peewee can create indexes on single or multiple columns, optionally including a *UNIQUE* constraint. Peewee also supports user-defined constraints on both models and fields.

Single-column indexes and constraints

Single column indexes are defined using field initialization parameters. The following example adds a unique index on the *username* field, and a normal index on the *email* field:

```
class User(Model):
    username = CharField(unique=True)
    email = CharField(index=True)
```

To add a user-defined constraint on a column, you can pass it in using the `constraints` parameter. You may wish to specify a default value as part of the schema, or add a `CHECK` constraint, for example:

```
class Product(Model):
    name = CharField(unique=True)
    price = DecimalField(constraints=[Check('price < 10000')])
    created = DateTimeField(
        constraints=[SQL("DEFAULT (datetime('now'))")])
```

Multi-column indexes

Multi-column indexes may be defined as *Meta* attributes using a nested tuple. Each database index is a 2-tuple, the first part of which is a tuple of the names of the fields, the second part a boolean indicating whether the index should be unique.

```
class Transaction(Model):
    from_acct = CharField()
    to_acct = CharField()
    amount = DecimalField()
    date = DateTimeField()

    class Meta:
        indexes = (
            # create a unique on from/to/date
            (('from_acct', 'to_acct', 'date'), True),

            # create a non-unique on from/to
```

(continues on next page)

(continued from previous page)

```
            (('from_acct', 'to_acct'), False),
        )
```

Note: Remember to add a **trailing comma** if your tuple of indexes contains only one item:

```
class Meta:
    indexes = (
        (('first_name', 'last_name'), True),  # Note the trailing comma!
    )
```

Advanced Index Creation

Peewee supports a more structured API for declaring indexes on a model using the *Model.add_index()* method or by directly using the *ModelIndex* helper class.

Examples:

```
class Article(Model):
    name = TextField()
    timestamp = TimestampField()
    status = IntegerField()
    flags = IntegerField()

# Add an index on "name" and "timestamp" columns.
Article.add_index(Article.name, Article.timestamp)

# Add a partial index on name and timestamp where status = 1.
Article.add_index(Article.name, Article.timestamp,
                  where=(Article.status == 1))

# Create a unique index on timestamp desc, status & 4.
idx = Article.index(
    Article.timestamp.desc(),
    Article.flags.bin_and(4),
    unique=True)
Article.add_index(idx)
```

Warning: SQLite does not support parameterized `CREATE INDEX` queries. This means that when using SQLite to create an index that involves an expression or scalar value, you will need to declare the index using the *SQL* helper:

```
# SQLite does not support parameterized CREATE INDEX queries, so
# we declare it manually.
Article.add_index(SQL('CREATE INDEX ...'))
```

See *add_index()* for details.

For more information, see:

- *Model.add_index()*
- *Model.index()*

- *ModelIndex*
- *Index*

Table constraints

Peewee allows you to add arbitrary constraints to your *Model*, that will be part of the table definition when the schema is created.

For instance, suppose you have a *people* table with a composite primary key of two columns, the person's first and last name. You wish to have another table relate to the *people* table, and to do this, you will need to define a foreign key constraint:

```
class Person(Model):
    first = CharField()
    last = CharField()

    class Meta:
        primary_key = CompositeKey('first', 'last')

class Pet(Model):
    owner_first = CharField()
    owner_last = CharField()
    pet_name = CharField()

    class Meta:
        constraints = [SQL('FOREIGN KEY(owner_first, owner_last) '
                           'REFERENCES person(first, last)')]
```

You can also implement `CHECK` constraints at the table level:

```
class Product(Model):
    name = CharField(unique=True)
    price = DecimalField()

    class Meta:
        constraints = [Check('price < 10000')]
```

1.7.5 Primary Keys, Composite Keys and other Tricks

The *AutoField* is used to identify an auto-incrementing integer primary key. If you do not specify a primary key, Peewee will automatically create an auto-incrementing primary key named "id".

To specify an auto-incrementing ID using a different field name, you can write:

```
class Event(Model):
    event_id = AutoField()  # Event.event_id will be auto-incrementing PK.
    name = CharField()
    timestamp = DateTimeField(default=datetime.datetime.now)
    metadata = BlobField()
```

You can identify a different field as the primary key, in which case an "id" column will not be created. In this example we will use a person's email address as the primary key:

```
class Person(Model):
    email = CharField(primary_key=True)
    name = TextField()
    dob = DateField()
```

Warning: I frequently see people write the following, expecting an auto-incrementing integer primary key:

```
class MyModel(Model):
    id = IntegerField(primary_key=True)
```

Peewee understands the above model declaration as a model with an integer primary key, but the value of that ID is determined by the application. To create an auto-incrementing integer primary key, you would instead write:

```
class MyModel(Model):
    id = AutoField()  # primary_key=True is implied.
```

Composite primary keys can be declared using *CompositeKey*. Note that doing this may cause issues with *ForeignKeyField*, as Peewee does not support the concept of a "composite foreign-key". As such, I've found it only advisable to use composite primary keys in a handful of situations, such as trivial many-to-many junction tables:

```
class Image(Model):
    filename = TextField()
    mimetype = CharField()

class Tag(Model):
    label = CharField()

class ImageTag(Model):  # Many-to-many relationship.
    image = ForeignKeyField(Image)
    tag = ForeignKeyField(Tag)

    class Meta:
        primary_key = CompositeKey('image', 'tag')
```

In the extremely rare case you wish to declare a model with *no* primary key, you can specify `primary_key = False` in the model `Meta` options.

Non-integer primary keys

If you would like use a non-integer primary key (which I generally don't recommend), you can specify `primary_key=True` when creating a field. When you wish to create a new instance for a model using a non-autoincrementing primary key, you need to be sure you *save()* specifying `force_insert=True`.

```
from peewee import *

class UUIDModel(Model):
    id = UUIDField(primary_key=True)
```

Auto-incrementing IDs are, as their name says, automatically generated for you when you insert a new row into the database. When you call *save()*, peewee determines whether to do an *INSERT* versus an *UPDATE* based on the presence of a primary key value. Since, with our uuid example, the database driver won't generate a new ID, we need to specify it manually. When we call save() for the first time, pass in `force_insert = True`:

```
# This works because .create() will specify `force_insert=True`.
obj1 = UUIDModel.create(id=uuid.uuid4())

# This will not work, however. Peewee will attempt to do an update:
obj2 = UUIDModel(id=uuid.uuid4())
obj2.save() # WRONG

obj2.save(force_insert=True) # CORRECT

# Once the object has been created, you can call save() normally.
obj2.save()
```

Note: Any foreign keys to a model with a non-integer primary key will have a `ForeignKeyField` use the same underlying storage type as the primary key they are related to.

Composite primary keys

Peewee has very basic support for composite keys. In order to use a composite key, you must set the `primary_key` attribute of the model options to a *CompositeKey* instance:

```
class BlogToTag(Model):
    """A simple "through" table for many-to-many relationship."""
    blog = ForeignKeyField(Blog)
    tag = ForeignKeyField(Tag)

    class Meta:
        primary_key = CompositeKey('blog', 'tag')
```

Warning: Peewee does not support foreign-keys to models that define a *CompositeKey* primary key. If you wish to add a foreign-key to a model that has a composite primary key, replicate the columns on the related model and add a custom accessor (e.g. a property).

Manually specifying primary keys

Sometimes you do not want the database to automatically generate a value for the primary key, for instance when bulk loading relational data. To handle this on a *one-off* basis, you can simply tell peewee to turn off `auto_increment` during the import:

```
data = load_user_csv() # load up a bunch of data

User._meta.auto_increment = False # turn off auto incrementing IDs
with db.atomic():
    for row in data:
        u = User(id=row[0], username=row[1])
        u.save(force_insert=True) # <-- force peewee to insert row

User._meta.auto_increment = True
```

Although a better way to accomplish the above, without resorting to hacks, is to use the *Model.insert_many()* API:

```
data = load_user_csv()
fields = [User.id, User.username]
with db.atomic():
    User.insert_many(data, fields=fields).execute()
```

If you *always* want to have control over the primary key, simply do not use the *AutoField* field type, but use a normal *IntegerField* (or other column type):

```
class User(BaseModel):
    id = IntegerField(primary_key=True)
    username = CharField()

>>> u = User.create(id=999, username='somebody')
>>> u.id
999
>>> User.get(User.username == 'somebody').id
999
```

Models without a Primary Key

If you wish to create a model with no primary key, you can specify `primary_key = False` in the inner `Meta` class:

```
class MyData(BaseModel):
    timestamp = DateTimeField()
    value = IntegerField()

    class Meta:
        primary_key = False
```

This will yield the following DDL:

```
CREATE TABLE "mydata" (
  "timestamp" DATETIME NOT NULL,
  "value" INTEGER NOT NULL
)
```

Warning: Some model APIs may not work correctly for models without a primary key, for instance *save()* and *delete_instance()* (you can instead use *insert()*, *update()* and *delete()*).

1.7.6 Self-referential foreign keys

When creating a hierarchical structure it is necessary to create a self-referential foreign key which links a child object to its parent. Because the model class is not defined at the time you instantiate the self-referential foreign key, use the special string `'self'` to indicate a self-referential foreign key:

```
class Category(Model):
    name = CharField()
    parent = ForeignKeyField('self', null=True, backref='children')
```

As you can see, the foreign key points *upward* to the parent object and the back-reference is named *children*.

Attention: Self-referential foreign-keys should always be `null=True`.

When querying against a model that contains a self-referential foreign key you may sometimes need to perform a self-join. In those cases you can use *`Model.alias()`* to create a table reference. Here is how you might query the category and parent model using a self-join:

```
Parent = Category.alias()
GrandParent = Category.alias()
query = (Category
         .select(Category, Parent)
         .join(Parent, on=(Category.parent == Parent.id))
         .join(GrandParent, on=(Parent.parent == GrandParent.id))
         .where(GrandParent.name == 'some category')
         .order_by(Category.name))
```

1.7.7 Circular foreign key dependencies

Sometimes it happens that you will create a circular dependency between two tables.

Note: My personal opinion is that circular foreign keys are a code smell and should be refactored (by adding an intermediary table, for instance).

Adding circular foreign keys with peewee is a bit tricky because at the time you are defining either foreign key, the model it points to will not have been defined yet, causing a `NameError`.

```
class User(Model):
    username = CharField()
    favorite_tweet = ForeignKeyField(Tweet, null=True)  # NameError!!

class Tweet(Model):
    message = TextField()
    user = ForeignKeyField(User, backref='tweets')
```

One option is to simply use an *`IntegerField`* to store the raw ID:

```
class User(Model):
    username = CharField()
    favorite_tweet_id = IntegerField(null=True)
```

By using *`DeferredForeignKey`* we can get around the problem and still use a foreign key field:

```
class User(Model):
    username = CharField()
    # Tweet has not been defined yet so use the deferred reference.
    favorite_tweet = DeferredForeignKey('Tweet', null=True)

class Tweet(Model):
    message = TextField()
    user = ForeignKeyField(User, backref='tweets')

# Now that Tweet is defined, "favorite_tweet" has been converted into
# a ForeignKeyField.
```

(continues on next page)

(continued from previous page)

```
print(User.favorite_tweet)
# <ForeignKeyField: "user"."favorite_tweet">
```

There is one more quirk to watch out for, though. When you call *create_table* we will again encounter the same issue. For this reason peewee will not automatically create a foreign key constraint for any *deferred* foreign keys.

To create the tables *and* the foreign-key constraint, you can use the *SchemaManager.create_foreign_key()* method to create the constraint after creating the tables:

```
# Will create the User and Tweet tables, but does *not* create a
# foreign-key constraint on User.favorite_tweet.
db.create_tables([User, Tweet])

# Create the foreign-key constraint:
User._schema.create_foreign_key(User.favorite_tweet)
```

Note: Because SQLite has limited support for altering tables, foreign-key constraints cannot be added to a table after it has been created.

1.8 Querying

This section will cover the basic CRUD operations commonly performed on a relational database:

- *Model.create()*, for executing *INSERT* queries.
- *Model.save()* and *Model.update()*, for executing *UPDATE* queries.
- *Model.delete_instance()* and *Model.delete()*, for executing *DELETE* queries.
- *Model.select()*, for executing *SELECT* queries.

Note: There is also a large collection of example queries taken from the Postgresql Exercises website. Examples are listed on the *query examples* document.

1.8.1 Creating a new record

You can use *Model.create()* to create a new model instance. This method accepts keyword arguments, where the keys correspond to the names of the model's fields. A new instance is returned and a row is added to the table.

```
>>> User.create(username='Charlie')
<__main__.User object at 0x2529350>
```

This will *INSERT* a new row into the database. The primary key will automatically be retrieved and stored on the model instance.

Alternatively, you can build up a model instance programmatically and then call *save()*:

```
>>> user = User(username='Charlie')
>>> user.save()  # save() returns the number of rows modified.
1
>>> user.id
```

(continues on next page)

(continued from previous page)

```
1
>>> huey = User()
>>> huey.username = 'Huey'
>>> huey.save()
1
>>> huey.id
2
```

When a model has a foreign key, you can directly assign a model instance to the foreign key field when creating a new record.

```
>>> tweet = Tweet.create(user=huey, message='Hello!')
```

You can also use the value of the related object's primary key:

```
>>> tweet = Tweet.create(user=2, message='Hello again!')
```

If you simply wish to insert data and do not need to create a model instance, you can use *Model.insert()*:

```
>>> User.insert(username='Mickey').execute()
3
```

After executing the insert query, the primary key of the new row is returned.

Note: There are several ways you can speed up bulk insert operations. Check out the *Bulk inserts* recipe section for more information.

1.8.2 Bulk inserts

There are a couple of ways you can load lots of data quickly. The naive approach is to simply call *Model.create()* in a loop:

```
data_source = [
    {'field1': 'val1-1', 'field2': 'val1-2'},
    {'field1': 'val2-1', 'field2': 'val2-2'},
    # ...
]

for data_dict in data_source:
    MyModel.create(**data_dict)
```

The above approach is slow for a couple of reasons:

1. If you are not wrapping the loop in a transaction then each call to *create()* happens in its own transaction. That is going to be really slow!
2. There is a decent amount of Python logic getting in your way, and each `InsertQuery` must be generated and parsed into SQL.
3. That's a lot of data (in terms of raw bytes of SQL) you are sending to your database to parse.
4. We are retrieving the *last insert id*, which causes an additional query to be executed in some cases.

You can get a significant speedup by simply wrapping this in a transaction with *atomic()*.

```
# This is much faster.
with db.atomic():
    for data_dict in data_source:
        MyModel.create(**data_dict)
```

The above code still suffers from points 2, 3 and 4. We can get another big boost by using *insert_many()*. This method accepts a list of tuples or dictionaries, and inserts multiple rows in a single query:

```
data_source = [
    {'field1': 'val1-1', 'field2': 'val1-2'},
    {'field1': 'val2-1', 'field2': 'val2-2'},
    # ...
]

# Fastest way to INSERT multiple rows.
MyModel.insert_many(data_source).execute()
```

The *insert_many()* method also accepts a list of row-tuples, provided you also specify the corresponding fields:

```
# We can INSERT tuples as well...
data = [('val1-1', 'val1-2'),
        ('val2-1', 'val2-2'),
        ('val3-1', 'val3-2')]

# But we need to indicate which fields the values correspond to.
MyModel.insert_many(data, fields=[MyModel.field1, MyModel.field2]).execute()
```

It is also a good practice to wrap the bulk insert in a transaction:

```
# You can, of course, wrap this in a transaction as well:
with db.atomic():
    MyModel.insert_many(data, fields=fields).execute()
```

Note: SQLite users should be aware of some caveats when using bulk inserts. Specifically, your SQLite3 version must be 3.7.11.0 or newer to take advantage of the bulk insert API. Additionally, by default SQLite limits the number of bound variables in a SQL query to `999`.

Inserting rows in batches

Depending on the number of rows in your data source, you may need to break it up into chunks. SQLite in particular typically has a limit of 999 variables-per-query (batch size would then be roughly 1000 / row length).

You can write a loop to batch your data into chunks (in which case it is **strongly recommended** you use a transaction):

```
# Insert rows 100 at a time.
with db.atomic():
    for idx in range(0, len(data_source), 100):
        MyModel.insert_many(data_source[idx:idx+100]).execute()
```

Peewee comes with a *chunked()* helper function which you can use for *efficiently* chunking a generic iterable into a series of *batch*-sized iterables:

```
from peewee import chunked

```

(continues on next page)

(continued from previous page)

```
# Insert rows 100 at a time.
with db.atomic():
    for batch in chunked(data_source, 100):
        MyModel.insert_many(batch).execute()
```

Alternatives

The *Model.bulk_create()* method behaves much like *Model.insert_many()*, but instead it accepts a list of unsaved model instances to insert, and it optionally accepts a batch-size parameter. To use the *bulk_create()* API:

```
# Read list of usernames from a file, for example.
with open('user_list.txt') as fh:
    # Create a list of unsaved User instances.
    users = [User(username=line.strip()) for line in fh.readlines()]

# Wrap the operation in a transaction and batch INSERT the users
# 100 at a time.
with db.atomic():
    User.bulk_create(users, batch_size=100)
```

Note: If you are using Postgresql (which supports the `RETURNING` clause), then the previously-unsaved model instances will have their new primary key values automatically populated.

In addition, Peewee also offers *Model.bulk_update()*, which can efficiently update one or more columns on a list of models. For example:

```
# First, create 3 users with usernames u1, u2, u3.
u1, u2, u3 = [User.create(username='u%s' % i) for i in (1, 2, 3)]

# Now we'll modify the user instances.
u1.username = 'u1-x'
u2.username = 'u2-y'
u3.username = 'u3-z'

# Update all three users with a single UPDATE query.
User.bulk_update([u1, u2, u3], fields=[User.username])
```

Note: For large lists of objects, you should specify a reasonable batch_size and wrap the call to *bulk_update()* with *Database.atomic()*:

```
with database.atomic():
    User.bulk_update(list_of_users, fields=['username'], batch_size=50)
```

Alternatively, you can use the *Database.batch_commit()* helper to process chunks of rows inside *batch*-sized transactions. This method also provides a workaround for databases besides Postgresql, when the primary-key of the newly-created rows must be obtained.

```
# List of row data to insert.
row_data = [{'username': 'u1'}, {'username': 'u2'}, ...]
```

(continues on next page)

(continued from previous page)

```
# Assume there are 789 items in row_data. The following code will result in
# 8 total transactions (7x100 rows + 1x89 rows).
for row in db.batch_commit(row_data, 100):
    User.create(**row)
```

Bulk-loading from another table

If the data you would like to bulk load is stored in another table, you can also create *INSERT* queries whose source is a *SELECT* query. Use the *Model.insert_from()* method:

```
res = (TweetArchive
       .insert_from(
           Tweet.select(Tweet.user, Tweet.message),
           fields=[TweetArchive.user, TweetArchive.message])
       .execute())
```

The above query is equivalent to the following SQL:

```
INSERT INTO "tweet_archive" ("user_id", "message")
SELECT "user_id", "message" FROM "tweet";
```

1.8.3 Updating existing records

Once a model instance has a primary key, any subsequent call to *save()* will result in an *UPDATE* rather than another *INSERT*. The model's primary key will not change:

```
>>> user.save()  # save() returns the number of rows modified.
1
>>> user.id
1
>>> user.save()
>>> user.id
1
>>> huey.save()
1
>>> huey.id
2
```

If you want to update multiple records, issue an *UPDATE* query. The following example will update all `Tweet` objects, marking them as *published*, if they were created before today. *Model.update()* accepts keyword arguments where the keys correspond to the model's field names:

```
>>> today = datetime.today()
>>> query = Tweet.update(is_published=True).where(Tweet.creation_date < today)
>>> query.execute()  # Returns the number of rows that were updated.
4
```

For more information, see the documentation on *Model.update()*, *Update* and *Model.bulk_update()*.

Note: If you would like more information on performing atomic updates (such as incrementing the value of a column), check out the *atomic update* recipes.

1.8.4 Atomic updates

Peewee allows you to perform atomic updates. Let's suppose we need to update some counters. The naive approach would be to write something like this:

```
>>> for stat in Stat.select().where(Stat.url == request.url):
...     stat.counter += 1
...     stat.save()
```

Do not do this! Not only is this slow, but it is also vulnerable to race conditions if multiple processes are updating the counter at the same time.

Instead, you can update the counters atomically using *update()*:

```
>>> query = Stat.update(counter=Stat.counter + 1).where(Stat.url == request.url)
>>> query.execute()
```

You can make these update statements as complex as you like. Let's give all our employees a bonus equal to their previous bonus plus 10% of their salary:

```
>>> query = Employee.update(bonus=(Employee.bonus + (Employee.salary * .1)))
>>> query.execute()  # Give everyone a bonus!
```

We can even use a subquery to update the value of a column. Suppose we had a denormalized column on the `User` model that stored the number of tweets a user had made, and we updated this value periodically. Here is how you might write such a query:

```
>>> subquery = Tweet.select(fn.COUNT(Tweet.id)).where(Tweet.user == User.id)
>>> update = User.update(num_tweets=subquery)
>>> update.execute()
```

Upsert

Peewee provides support for varying types of upsert functionality. With SQLite prior to 3.24.0 and MySQL, Peewee offers the *replace()*, which allows you to insert a record or, in the event of a constraint violation, replace the existing record.

Example of using *replace()* and *on_conflict_replace()*:

```
class User(Model):
    username = TextField(unique=True)
    last_login = DateTimeField(null=True)

# Insert or update the user. The "last_login" value will be updated
# regardless of whether the user existed previously.
user_id = (User
           .replace(username='the-user', last_login=datetime.now())
           .execute())

# This query is equivalent:
user_id = (User
           .insert(username='the-user', last_login=datetime.now())
           .on_conflict_replace()
           .execute())
```

Note: In addition to *replace*, SQLite, MySQL and Postgresql provide an *ignore* action (see: *on_conflict_ignore()*) if you simply wish to insert and ignore any potential constraint violation.

MySQL supports upsert via the *ON DUPLICATE KEY UPDATE* clause. For example:

```
class User(Model):
    username = TextField(unique=True)
    last_login = DateTimeField(null=True)
    login_count = IntegerField()

# Insert a new user.
User.create(username='huey', login_count=0)

# Simulate the user logging in. The login count and timestamp will be
# either created or updated correctly.
now = datetime.now()
rowid = (User
         .insert(username='huey', last_login=now, login_count=1)
         .on_conflict(
             preserve=[User.last_login],  # Use the value we would have inserted.
             update={User.login_count: User.login_count + 1})
         .execute())
```

In the above example, we could safely invoke the upsert query as many times as we wanted. The login count will be incremented atomically, the last login column will be updated, and no duplicate rows will be created.

Postgresql and SQLite (3.24.0 and newer) provide a different syntax that allows for more granular control over which constraint violation should trigger the conflict resolution, and what values should be updated or preserved.

Example of using *on_conflict()* to perform a Postgresql-style upsert (or SQLite 3.24+):

```
class User(Model):
    username = TextField(unique=True)
    last_login = DateTimeField(null=True)
    login_count = IntegerField()

# Insert a new user.
User.create(username='huey', login_count=0)

# Simulate the user logging in. The login count and timestamp will be
# either created or updated correctly.
now = datetime.now()
rowid = (User
         .insert(username='huey', last_login=now, login_count=1)
         .on_conflict(
             conflict_target=[User.username],  # Which constraint?
             preserve=[User.last_login],  # Use the value we would have inserted.
             update={User.login_count: User.login_count + 1})
         .execute())
```

In the above example, we could safely invoke the upsert query as many times as we wanted. The login count will be incremented atomically, the last login column will be updated, and no duplicate rows will be created.

Note: The main difference between MySQL and Postgresql/SQLite is that Postgresql and SQLite require that you specify a `conflict_target`.

Here is a more advanced (if contrived) example using the *EXCLUDED* namespace. The *EXCLUDED* helper allows us to reference values in the conflicting data. For our example, we'll assume a simple table mapping a unique key (string) to a value (integer):

```
class KV(Model):
    key = CharField(unique=True)
    value = IntegerField()

# Create one row.
KV.create(key='k1', value=1)

# Demonstrate usage of EXCLUDED.
# Here we will attempt to insert a new value for a given key. If that
# key already exists, then we will update its value with the *sum* of its
# original value and the value we attempted to insert -- provided that
# the new value is larger than the original value.
query = (KV.insert(key='k1', value=10)
         .on_conflict(conflict_target=[KV.key],
                      update={KV.value: KV.value + EXCLUDED.value},
                      where=(EXCLUDED.value > KV.value)))

# Executing the above query will result in the following data being
# present in the "kv" table:
# (key='k1', value=11)
query.execute()

# If we attempted to execute the query *again*, then nothing would be
# updated, as the new value (10) is now less than the value in the
# original row (11).
```

For more information, see *Insert.on_conflict()* and *OnConflict*.

1.8.5 Deleting records

To delete a single model instance, you can use the *Model.delete_instance()* shortcut. *delete_instance()* will delete the given model instance and can optionally delete any dependent objects recursively (by specifying *recursive=True*).

```
>>> user = User.get(User.id == 1)
>>> user.delete_instance()  # Returns the number of rows deleted.
1

>>> User.get(User.id == 1)
UserDoesNotExist: instance matching query does not exist:
SQL: SELECT t1."id", t1."username" FROM "user" AS t1 WHERE t1."id" = ?
PARAMS: [1]
```

To delete an arbitrary set of rows, you can issue a *DELETE* query. The following will delete all `Tweet` objects that are over one year old:

```
>>> query = Tweet.delete().where(Tweet.creation_date < one_year_ago)
>>> query.execute()  # Returns the number of rows deleted.
7
```

For more information, see the documentation on:

- *Model.delete_instance()*

- *Model.delete()*
- DeleteQuery

1.8.6 Selecting a single record

You can use the *Model.get()* method to retrieve a single instance matching the given query. For primary-key lookups, you can also use the shortcut method *Model.get_by_id()*.

This method is a shortcut that calls *Model.select()* with the given query, but limits the result set to a single row. Additionally, if no model matches the given query, a DoesNotExist exception will be raised.

```
>>> User.get(User.id == 1)
<__main__.User object at 0x25294d0>

>>> User.get_by_id(1)  # Same as above.
<__main__.User object at 0x252df10>

>>> User[1]  # Also same as above.
<__main__.User object at 0x252dd10>

>>> User.get(User.id == 1).username
u'Charlie'

>>> User.get(User.username == 'Charlie')
<__main__.User object at 0x2529410>

>>> User.get(User.username == 'nobody')
UserDoesNotExist: instance matching query does not exist:
SQL: SELECT t1."id", t1."username" FROM "user" AS t1 WHERE t1."username" = ?
PARAMS: ['nobody']
```

For more advanced operations, you can use *SelectBase.get()*. The following query retrieves the latest tweet from the user named *charlie*:

```
>>> (Tweet
...  .select()
...  .join(User)
...  .where(User.username == 'charlie')
...  .order_by(Tweet.created_date.desc())
...  .get())
<__main__.Tweet object at 0x2623410>
```

For more information, see the documentation on:

- *Model.get()*
- *Model.get_by_id()*
- *Model.get_or_none()* - if no matching row is found, return None.
- Model.first()
- *Model.select()*
- *SelectBase.get()*

1.8.7 Create or get

Peewee has one helper method for performing "get/create" type operations: *Model.get_or_create()*, which first attempts to retrieve the matching row. Failing that, a new row will be created.

For "create or get" type logic, typically one would rely on a *unique* constraint or primary key to prevent the creation of duplicate objects. As an example, let's say we wish to implement registering a new user account using the *example User model*. The *User* model has a *unique* constraint on the username field, so we will rely on the database's integrity guarantees to ensure we don't end up with duplicate usernames:

```
try:
    with db.atomic():
        return User.create(username=username)
except peewee.IntegrityError:
    # `username` is a unique column, so this username already exists,
    # making it safe to call .get().
    return User.get(User.username == username)
```

You can easily encapsulate this type of logic as a `classmethod` on your own `Model` classes.

The above example first attempts at creation, then falls back to retrieval, relying on the database to enforce a unique constraint. If you prefer to attempt to retrieve the record first, you can use *get_or_create()*. This method is implemented along the same lines as the Django function of the same name. You can use the Django-style keyword argument filters to specify your `WHERE` conditions. The function returns a 2-tuple containing the instance and a boolean value indicating if the object was created.

Here is how you might implement user account creation using *get_or_create()*:

```
user, created = User.get_or_create(username=username)
```

Suppose we have a different model `Person` and would like to get or create a person object. The only conditions we care about when retrieving the `Person` are their first and last names, **but** if we end up needing to create a new record, we will also specify their date-of-birth and favorite color:

```
person, created = Person.get_or_create(
    first_name=first_name,
    last_name=last_name,
    defaults={'dob': dob, 'favorite_color': 'green'})
```

Any keyword argument passed to *get_or_create()* will be used in the `get()` portion of the logic, except for the `defaults` dictionary, which will be used to populate values on newly-created instances.

For more details read the documentation for *Model.get_or_create()*.

1.8.8 Selecting multiple records

We can use *Model.select()* to retrieve rows from the table. When you construct a *SELECT* query, the database will return any rows that correspond to your query. Peewee allows you to iterate over these rows, as well as use indexing and slicing operations:

```
>>> query = User.select()
>>> [user.username for user in query]
['Charlie', 'Huey', 'Peewee']

>>> query[1]
<__main__.User at 0x7f83e80f5550>
```

(continues on next page)

(continued from previous page)

```
>>> query[1].username
'Huey'

>>> query[:2]
[<__main__.User at 0x7f83e80f53a8>, <__main__.User at 0x7f83e80f5550>]
```

Select queries are smart, in that you can iterate, index and slice the query multiple times but the query is only executed once.

In the following example, we will simply call *select()* and iterate over the return value, which is an instance of *Select*. This will return all the rows in the *User* table:

```
>>> for user in User.select():
...     print user.username
...
Charlie
Huey
Peewee
```

Note: Subsequent iterations of the same query will not hit the database as the results are cached. To disable this behavior (to reduce memory usage), call `Select.iterator()` when iterating.

When iterating over a model that contains a foreign key, be careful with the way you access values on related models. Accidentally resolving a foreign key or iterating over a back-reference can cause *N+1 query behavior*.

When you create a foreign key, such as `Tweet.user`, you can use the *backref* to create a back-reference (`User.tweets`). Back-references are exposed as *Select* instances:

```
>>> tweet = Tweet.get()
>>> tweet.user  # Accessing a foreign key returns the related model.
<tw.User at 0x7f3ceb017f50>

>>> user = User.get()
>>> user.tweets  # Accessing a back-reference returns a query.
<peewee.ModelSelect at 0x7f73db3bafd0>
```

You can iterate over the `user.tweets` back-reference just like any other *Select*:

```
>>> for tweet in user.tweets:
...     print(tweet.message)
...
hello world
this is fun
look at this picture of my food
```

In addition to returning model instances, *Select* queries can return dictionaries, tuples and namedtuples. Depending on your use-case, you may find it easier to work with rows as dictionaries, for example:

```
>>> query = User.select().dicts()
>>> for row in query:
...     print(row)

{'id': 1, 'username': 'Charlie'}
{'id': 2, 'username': 'Huey'}
{'id': 3, 'username': 'Peewee'}
```

See *namedtuples()*, *tuples()*, *dicts()* for more information.

Iterating over large result-sets

By default peewee will cache the rows returned when iterating over a *Select* query. This is an optimization to allow multiple iterations as well as indexing and slicing without causing additional queries. This caching can be problematic, however, when you plan to iterate over a large number of rows.

To reduce the amount of memory used by peewee when iterating over a query, use the *iterator()* method. This method allows you to iterate without caching each model returned, using much less memory when iterating over large result sets.

```
# Let's assume we've got 10 million stat objects to dump to a csv file.
stats = Stat.select()

# Our imaginary serializer class
serializer = CSVSerializer()

# Loop over all the stats and serialize.
for stat in stats.iterator():
    serializer.serialize_object(stat)
```

For simple queries you can see further speed improvements by returning rows as dictionaries, namedtuples or tuples. The following methods can be used on any *Select* query to change the result row type:

- *dicts()*
- *namedtuples()*
- *tuples()*

Don't forget to append the *iterator()* method call to also reduce memory consumption. For example, the above code might look like:

```
# Let's assume we've got 10 million stat objects to dump to a csv file.
stats = Stat.select()

# Our imaginary serializer class
serializer = CSVSerializer()

# Loop over all the stats (rendered as tuples, without caching) and serialize.
for stat_tuple in stats.tuples().iterator():
    serializer.serialize_tuple(stat_tuple)
```

When iterating over a large number of rows that contain columns from multiple tables, peewee will reconstruct the model graph for each row returned. This operation can be slow for complex graphs. For example, if we were selecting a list of tweets along with the username and avatar of the tweet's author, Peewee would have to create two objects for each row (a tweet and a user). In addition to the above row-types, there is a fourth method *objects()* which will return the rows as model instances, but will not attempt to resolve the model graph.

For example:

```
query = (Tweet
         .select(Tweet, User)  # Select tweet and user data.
         .join(User))

# Note that the user columns are stored in a separate User instance
# accessible at tweet.user:
```

(continues on next page)

(continued from previous page)

```
for tweet in query:
    print(tweet.user.username, tweet.content)

# Using ".objects()" will not create the tweet.user object and assigns all
# user attributes to the tweet instance:
for tweet in query.objects():
    print(tweet.username, tweet.content)
```

For maximum performance, you can execute queries and then iterate over the results using the underlying database cursor. *Database.execute()* accepts a query object, executes the query, and returns a DB-API 2.0 `Cursor` object. The cursor will return the raw row-tuples:

```
query = Tweet.select(Tweet.content, User.username).join(User)
cursor = database.execute(query)
for (content, username) in cursor:
    print(username, '->', content)
```

1.8.9 Filtering records

You can filter for particular records using normal python operators. Peewee supports a wide variety of *query operators*.

```
>>> user = User.get(User.username == 'Charlie')
>>> for tweet in Tweet.select().where(Tweet.user == user, Tweet.is_published == True):
...     print(tweet.user.username, '->', tweet.message)
...
Charlie -> hello world
Charlie -> this is fun

>>> for tweet in Tweet.select().where(Tweet.created_date < datetime.datetime(2011, 1,␣
↪1)):
...     print(tweet.message, tweet.created_date)
...
Really old tweet 2010-01-01 00:00:00
```

You can also filter across joins:

```
>>> for tweet in Tweet.select().join(User).where(User.username == 'Charlie'):
...     print(tweet.message)
hello world
this is fun
look at this picture of my food
```

If you want to express a complex query, use parentheses and python's bitwise *or* and *and* operators:

```
>>> Tweet.select().join(User).where(
...     (User.username == 'Charlie') |
...     (User.username == 'Peewee Herman'))
```

Note: Note that Peewee uses **bitwise** operators (`&` and `|`) rather than logical operators (`and` and `or`). The reason for this is that Python coerces the return value of logical operations to a boolean value. This is also the reason why "IN" queries must be expressed using `.in_()` rather than the `in` operator.

Check out *the table of query operations* to see what types of queries are possible.

Note: A lot of fun things can go in the where clause of a query, such as:

- A field expression, e.g. `User.username == 'Charlie'`
- A function expression, e.g. `fn.Lower(fn.Substr(User.username, 1, 1)) == 'a'`
- A comparison of one column to another, e.g. `Employee.salary < (Employee.tenure * 1000) + 40000`

You can also nest queries, for example tweets by users whose username starts with "a":

```
# get users whose username starts with "a"
a_users = User.select().where(fn.Lower(fn.Substr(User.username, 1, 1)) == 'a')

# the ".in_()" method signifies an "IN" query
a_user_tweets = Tweet.select().where(Tweet.user.in_(a_users))
```

More query examples

Note: For a wide range of example queries, see the *Query Examples* document, which shows how to implements queries from the PostgreSQL Exercises website.

Get active users:

```
User.select().where(User.active == True)
```

Get users who are either staff or superusers:

```
User.select().where(
    (User.is_staff == True) | (User.is_superuser == True))
```

Get tweets by user named "charlie":

```
Tweet.select().join(User).where(User.username == 'charlie')
```

Get tweets by staff or superusers (assumes FK relationship):

```
Tweet.select().join(User).where(
    (User.is_staff == True) | (User.is_superuser == True))
```

Get tweets by staff or superusers using a subquery:

```
staff_super = User.select(User.id).where(
    (User.is_staff == True) | (User.is_superuser == True))
Tweet.select().where(Tweet.user.in_(staff_super))
```

1.8.10 Sorting records

To return rows in order, use the *order_by()* method:

```
>>> for t in Tweet.select().order_by(Tweet.created_date):
...     print(t.pub_date)
...
2010-01-01 00:00:00
2011-06-07 14:08:48
2011-06-07 14:12:57

>>> for t in Tweet.select().order_by(Tweet.created_date.desc()):
...     print(t.pub_date)
...
2011-06-07 14:12:57
2011-06-07 14:08:48
2010-01-01 00:00:00
```

You can also use + and − prefix operators to indicate ordering:

```
# The following queries are equivalent:
Tweet.select().order_by(Tweet.created_date.desc())

Tweet.select().order_by(-Tweet.created_date)  # Note the "-" prefix.

# Similarly you can use "+" to indicate ascending order, though ascending
# is the default when no ordering is otherwise specified.
User.select().order_by(+User.username)
```

You can also order across joins. Assuming you want to order tweets by the username of the author, then by created_date:

```
query = (Tweet
         .select()
         .join(User)
         .order_by(User.username, Tweet.created_date.desc()))
```

```
SELECT t1."id", t1."user_id", t1."message", t1."is_published", t1."created_date"
FROM "tweet" AS t1
INNER JOIN "user" AS t2
  ON t1."user_id" = t2."id"
ORDER BY t2."username", t1."created_date" DESC
```

When sorting on a calculated value, you can either include the necessary SQL expressions, or reference the alias assigned to the value. Here are two examples illustrating these methods:

```
# Let's start with our base query. We want to get all usernames and the number of
# tweets they've made. We wish to sort this list from users with most tweets to
# users with fewest tweets.
query = (User
         .select(User.username, fn.COUNT(Tweet.id).alias('num_tweets'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User.username))
```

You can order using the same COUNT expression used in the `select` clause. In the example below we are ordering by the `COUNT()` of tweet ids descending:

```
query = (User
         .select(User.username, fn.COUNT(Tweet.id).alias('num_tweets'))
         .join(Tweet, JOIN.LEFT_OUTER)
```

(continues on next page)

(continued from previous page)

```
         .group_by(User.username)
         .order_by(fn.COUNT(Tweet.id).desc()))
```

Alternatively, you can reference the alias assigned to the calculated value in the `select` clause. This method has the benefit of being a bit easier to read. Note that we are not referring to the named alias directly, but are wrapping it using the *SQL* helper:

```
query = (User
         .select(User.username, fn.COUNT(Tweet.id).alias('num_tweets'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User.username)
         .order_by(SQL('num_tweets').desc()))
```

Or, to do things the "peewee" way:

```
ntweets = fn.COUNT(Tweet.id)
query = (User
         .select(User.username, ntweets.alias('num_tweets'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User.username)
         .order_by(ntweets.desc())
```

1.8.11 Getting random records

Occasionally you may want to pull a random record from the database. You can accomplish this by ordering by the *random* or *rand* function (depending on your database):

Postgresql and Sqlite use the *Random* function:

```
# Pick 5 lucky winners:
LotteryNumber.select().order_by(fn.Random()).limit(5)
```

MySQL uses *Rand*:

```
# Pick 5 lucky winners:
LotterNumber.select().order_by(fn.Rand()).limit(5)
```

1.8.12 Paginating records

The *paginate()* method makes it easy to grab a *page* or records. *paginate()* takes two parameters, `page_number`, and `items_per_page`.

Attention: Page numbers are 1-based, so the first page of results will be page 1.

```
>>> for tweet in Tweet.select().order_by(Tweet.id).paginate(2, 10):
...     print(tweet.message)
...
tweet 10
tweet 11
tweet 12
tweet 13
```

(continues on next page)

(continued from previous page)

```
tweet 14
tweet 15
tweet 16
tweet 17
tweet 18
tweet 19
```

If you would like more granular control, you can always use *limit()* and *offset()*.

1.8.13 Counting records

You can count the number of rows in any select query:

```
>>> Tweet.select().count()
100
>>> Tweet.select().where(Tweet.id > 50).count()
50
```

Peewee will wrap your query in an outer query that performs a count, which results in SQL like:

```
SELECT COUNT(1) FROM ( ... your query ... );
```

1.8.14 Aggregating records

Suppose you have some users and want to get a list of them along with the count of tweets in each.

```
query = (User
         .select(User, fn.Count(Tweet.id).alias('count'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User))
```

The resulting query will return *User* objects with all their normal attributes plus an additional attribute *count* which will contain the count of tweets for each user. We use a left outer join to include users who have no tweets.

Let's assume you have a tagging application and want to find tags that have a certain number of related objects. For this example we'll use some different models in a *many-to-many* configuration:

```
class Photo(Model):
    image = CharField()

class Tag(Model):
    name = CharField()

class PhotoTag(Model):
    photo = ForeignKeyField(Photo)
    tag = ForeignKeyField(Tag)
```

Now say we want to find tags that have at least 5 photos associated with them:

```
query = (Tag
         .select()
         .join(PhotoTag)
         .join(Photo)
```

(continues on next page)

(continued from previous page)

```
         .group_by(Tag)
         .having(fn.Count(Photo.id) > 5))
```

This query is equivalent to the following SQL:

```
SELECT t1."id", t1."name"
FROM "tag" AS t1
INNER JOIN "phototag" AS t2 ON t1."id" = t2."tag_id"
INNER JOIN "photo" AS t3 ON t2."photo_id" = t3."id"
GROUP BY t1."id", t1."name"
HAVING Count(t3."id") > 5
```

Suppose we want to grab the associated count and store it on the tag:

```
query = (Tag
         .select(Tag, fn.Count(Photo.id).alias('count'))
         .join(PhotoTag)
         .join(Photo)
         .group_by(Tag)
         .having(fn.Count(Photo.id) > 5))
```

1.8.15 Retrieving Scalar Values

You can retrieve scalar values by calling `Query.scalar()`. For instance:

```
>>> PageView.select(fn.Count(fn.Distinct(PageView.url))).scalar()
100
```

You can retrieve multiple scalar values by passing `as_tuple=True`:

```
>>> Employee.select(
...     fn.Min(Employee.salary), fn.Max(Employee.salary)
... ).scalar(as_tuple=True)
(30000, 50000)
```

1.8.16 Window functions

A *`Window`* function refers to an aggregate function that operates on a sliding window of data that is being processed as part of a `SELECT` query. Window functions make it possible to do things like:

1. Perform aggregations against subsets of a result-set.
2. Calculate a running total.
3. Rank results.
4. Compare a row value to a value in the preceding (or succeeding!) row(s).

peewee comes with support for SQL window functions, which can be created by calling *`Function.over()`* and passing in your partitioning or ordering parameters.

For the following examples, we'll use the following model and sample data:

```
class Sample(Model):
    counter = IntegerField()
    value = FloatField()

data = [(1, 10),
        (1, 20),
        (2, 1),
        (2, 3),
        (3, 100)]
Sample.insert_many(data, fields=[Sample.counter, Sample.value]).execute()
```

Our sample table now contains:

id	counter	value
1	1	10.0
2	1	20.0
3	2	1.0
4	2	3.0
5	3	100.0

Ordered Windows

Let's calculate a running sum of the `value` field. In order for it to be a "running" sum, we need it to be ordered, so we'll order with respect to the Sample's `id` field:

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.SUM(Sample.value).over(order_by=[Sample.id]).alias('total'))

for sample in query:
    print(sample.counter, sample.value, sample.total)

# 1    10.    10.
# 1    20.    30.
# 2     1.    31.
# 2     3.    34.
# 3   100    134.
```

For another example, we'll calculate the difference between the current value and the previous value, when ordered by the `id`:

```
difference = Sample.value - fn.LAG(Sample.value, 1).over(order_by=[Sample.id])
query = Sample.select(
    Sample.counter,
    Sample.value,
    difference.alias('diff'))

for sample in query:
    print(sample.counter, sample.value, sample.diff)

# 1    10.   NULL
# 1    20.    10.  -- (20 - 10)
# 2     1.   -19.  -- (1 - 20)
```

(continues on next page)

(continued from previous page)

```
# 2     3.     2.  -- (3 - 1)
# 3   100     97.  -- (100 - 3)
```

Partitioned Windows

Let's calculate the average `value` for each distinct "counter" value. Notice that there are three possible values for the `counter` field (1, 2, and 3). We can do this by calculating the `AVG()` of the `value` column over a window that is partitioned depending on the `counter` field:

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.AVG(Sample.value).over(partition_by=[Sample.counter]).alias('cavg'))

for sample in query:
    print(sample.counter, sample.value, sample.cavg)

# 1    10.    15.
# 1    20.    15.
# 2     1.     2.
# 2     3.     2.
# 3   100    100.
```

We can use ordering within partitions by specifying both the `order_by` and `partition_by` parameters. For an example, let's rank the samples by value within each distinct `counter` group.

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.RANK().over(
        order_by=[Sample.value],
        partition_by=[Sample.counter]).alias('rank'))

for sample in query:
    print(sample.counter, sample.value, sample.rank)

# 1    10.    1
# 1    20.    2
# 2     1.    1
# 2     3.    2
# 3   100     1
```

Bounded windows

By default, window functions are evaluated using an *unbounded preceding* start for the window, and the *current row* as the end. We can change the bounds of the window our aggregate functions operate on by specifying a `start` and/or `end` in the call to *`Function.over()`*. Additionally, Peewee comes with helper-methods on the *`Window`* object for generating the appropriate boundary references:

- *`Window.CURRENT_ROW`* - attribute that references the current row.
- *`Window.preceding()`* - specify number of row(s) preceding, or omit number to indicate **all** preceding rows.
- *`Window.following()`* - specify number of row(s) following, or omit number to indicate **all** following rows.

To examine how boundaries work, we'll calculate a running total of the `value` column, ordered with respect to `id`, **but** we'll only look the running total of the current row and it's two preceding rows:

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.SUM(Sample.value).over(
        order_by=[Sample.id],
        start=Window.preceding(2),
        end=Window.CURRENT_ROW).alias('rsum'))

for sample in query:
    print(sample.counter, sample.value, sample.rsum)

# 1    10.    10.
# 1    20.    30.  -- (20 + 10)
# 2     1.    31.  -- (1 + 20 + 10)
# 2     3.    24.  -- (3 + 1 + 20)
# 3   100    104.  -- (100 + 3 + 1)
```

Note: Technically we did not need to specify the `end=Window.CURRENT` because that is the default. It was shown in the example for demonstration.

Let's look at another example. In this example we will calculate the "opposite" of a running total, in which the total sum of all values is decreased by the value of the samples, ordered by `id`. To accomplish this, we'll calculate the sum from the current row to the last row.

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.SUM(Sample.value).over(
        order_by=[Sample.id],
        start=Window.CURRENT_ROW,
        end=Window.following()).alias('rsum'))

# 1    10.   134.  -- (10 + 20 + 1 + 3 + 100)
# 1    20.   124.  -- (20 + 1 + 3 + 100)
# 2     1.   104.  -- (1 + 3 + 100)
# 2     3.   103.  -- (3 + 100)
# 3   100    100.  -- (100)
```

Filtered Aggregates

Aggregate functions may also support filter functions (Postgres and Sqlite 3.25+), which get translated into a `FILTER (WHERE...)` clause. Filter expressions are added to an aggregate function with the *`Function.filter()`* method.

For an example, we will calculate the running sum of the `value` field with respect to the `id`, but we will filter-out any samples whose `counter=2`.

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.SUM(Sample.value).filter(Sample.counter != 2).over(
```

(continues on next page)

(continued from previous page)

```
        order_by=[Sample.id]).alias('csum'))

for sample in query:
    print(sample.counter, sample.value, sample.csum)

# 1    10.    10.
# 1    20.    30.
# 2     1.    30.
# 2     3.    30.
# 3   100    130.
```

Note: The call to *filter()* must precede the call to *over()*.

Reusing Window Definitions

If you intend to use the same window definition for multiple aggregates, you can create a *Window* object. The *Window* object takes the same parameters as *Function.over()*, and can be passed to the `over()` method in-place of the individual parameters.

Here we'll declare a single window, ordered with respect to the sample `id`, and call several window functions using that window definition:

```
win = Window(order_by=[Sample.id])
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.LEAD(Sample.value).over(win),
    fn.LAG(Sample.value).over(win),
    fn.SUM(Sample.value).over(win)
).window(win)  # Include our window definition in query.

for row in query.tuples():
    print(row)

# counter  value  lead()  lag()  sum()
# 1          10.     20.   NULL    10.
# 1          20.      1.    10.    30.
# 2           1.      3.    20.    31.
# 2           3.    100.     1.    34.
# 3         100.    NULL     3.   134.
```

Multiple window definitions

In the previous example, we saw how to declare a *Window* definition and re-use it for multiple different aggregations. You can include as many window definitions as you need in your queries, but it is necessary to ensure each window has a unique alias:

```
w1 = Window(order_by=[Sample.id]).alias('w1')
w2 = Window(partition_by=[Sample.counter]).alias('w2')
query = Sample.select(
    Sample.counter,
    Sample.value,
```

(continues on next page)

(continued from previous page)

```
    fn.SUM(Sample.value).over(w1).alias('rsum'),  # Running total.
    fn.AVG(Sample.value).over(w2).alias('cavg')   # Avg per category.
).window(w1, w2)  # Include our window definitions.

for sample in query:
    print(sample.counter, sample.value, sample.rsum, sample.cavg)

# counter  value   rsum     cavg
# 1          10.     10.     15.
# 1          20.     30.     15.
# 2           1.     31.      2.
# 2           3.     34.      2.
# 3         100     134.    100.
```

Similarly, if you have multiple window definitions that share similar definitions, it is possible to extend a previously-defined window definition. For example, here we will be partitioning the data-set by the counter value, so we'll be doing our aggregations with respect to the counter. Then we'll define a second window that extends this partitioning, and adds an ordering clause:

```
w1 = Window(partition_by=[Sample.counter]).alias('w1')

# By extending w1, this window definition will also be partitioned
# by "counter".
w2 = Window(extends=w1, order_by=[Sample.value.desc()]).alias('w2')

query = (Sample
         .select(Sample.counter, Sample.value,
                 fn.SUM(Sample.value).over(w1).alias('group_sum'),
                 fn.RANK().over(w2).alias('revrank'))
         .window(w1, w2)
         .order_by(Sample.id))

for sample in query:
    print(sample.counter, sample.value, sample.group_sum, sample.revrank)

# counter  value   group_sum   revrank
# 1        10.     30.         2
# 1        20.     30.         1
# 2        1.      4.          2
# 2        3.      4.          1
# 3        100.    100.        1
```

Frame types: RANGE vs ROWS vs GROUPS

Depending on the frame type, the database will process ordered groups differently. Let's create two additional `Sample` rows to visualize the difference:

```
>>> Sample.create(counter=1, value=20.)
<Sample 6>
>>> Sample.create(counter=2, value=1.)
<Sample 7>
```

Our table now contains:

id	counter	value
1	1	10.0
2	1	20.0
3	2	1.0
4	2	3.0
5	3	100.0
6	1	20.0
7	2	1.0

Let's examine the difference by calculating a "running sum" of the samples, ordered with respect to the `counter` and `value` fields. To specify the frame type, we can use either:

- *Window.RANGE*
- *Window.ROWS*
- *Window.GROUPS*

The behavior of *RANGE*, when there are logical duplicates, may lead to unexpected results:

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.SUM(Sample.value).over(
        order_by=[Sample.counter, Sample.value],
        frame_type=Window.RANGE).alias('rsum'))

for sample in query.order_by(Sample.counter, Sample.value):
    print(sample.counter, sample.value, sample.rsum)

# counter  value   rsum
# 1          10.     10.
# 1          20.     50.
# 1          20.     50.
# 2           1.     52.
# 2           1.     52.
# 2           3.     55.
# 3         100     155.
```

With the inclusion of the new rows we now have some rows that have duplicate `category` and `value` values. The *RANGE* frame type causes these duplicates to be evaluated together rather than separately.

The more expected result can be achieved by using *ROWS* as the frame-type:

```
query = Sample.select(
    Sample.counter,
    Sample.value,
    fn.SUM(Sample.value).over(
        order_by=[Sample.counter, Sample.value],
        frame_type=Window.ROWS).alias('rsum'))

for sample in query.order_by(Sample.counter, Sample.value):
    print(sample.counter, sample.value, sample.rsum)

# counter  value   rsum
# 1          10.     10.
# 1          20.     30.
```

(continues on next page)

(continued from previous page)

```
# 1          20.     50.
# 2           1.     51.
# 2           1.     52.
# 2           3.     55.
# 3         100     155.
```

Peewee uses these rules for determining what frame-type to use:

- If the user specifies a `frame_type`, that frame type will be used.
- If `start` and/or `end` boundaries are specified Peewee will default to using `ROWS`.
- If the user did not specify frame type or start/end boundaries, Peewee will use the database default, which is `RANGE`.

The *Window.GROUPS* frame type looks at the window range specification in terms of groups of rows, based on the ordering term(s). Using `GROUPS`, we can define the frame so it covers distinct groupings of rows. Let's look at an example:

```
query = (Sample
         .select(Sample.counter, Sample.value,
                 fn.SUM(Sample.value).over(
                    order_by=[Sample.counter, Sample.value],
                    frame_type=Window.GROUPS,
                    start=Window.preceding(1)).alias('gsum'))
         .order_by(Sample.counter, Sample.value))

for sample in query:
    print(sample.counter, sample.value, sample.gsum)

#  counter   value    gsum
#  1         10       10
#  1         20       50
#  1         20       50   (10) + (20+0)
#  2         1        42
#  2         1        42   (20+20) + (1+1)
#  2         3        5    (1+1) + 3
#  3         100      103  (3) + 100
```

As you can hopefully infer, the window is grouped by its ordering term, which is `(counter, value)`. We are looking at a window that extends between one previous group and the current group.

Note: For information about the window function APIs, see:

- *Function.over()*
- *Function.filter()*
- *Window*

For general information on window functions, read the postgres window functions tutorial

Additionally, the postgres docs and the sqlite docs contain a lot of good information.

1.8.17 Retrieving row tuples / dictionaries / namedtuples

Sometimes you do not need the overhead of creating model instances and simply want to iterate over the row data without needing all the APIs provided *Model*. To do this, use:

- *dicts()*
- *namedtuples()*
- *tuples()*
- *objects()* – accepts an arbitrary constructor function which is called with the row tuple.

```
stats = (Stat
         .select(Stat.url, fn.Count(Stat.url))
         .group_by(Stat.url)
         .tuples())

# iterate over a list of 2-tuples containing the url and count
for stat_url, stat_count in stats:
    print(stat_url, stat_count)
```

Similarly, you can return the rows from the cursor as dictionaries using *dicts()*:

```
stats = (Stat
         .select(Stat.url, fn.Count(Stat.url).alias('ct'))
         .group_by(Stat.url)
         .dicts())

# iterate over a list of 2-tuples containing the url and count
for stat in stats:
    print(stat['url'], stat['ct'])
```

1.8.18 Returning Clause

PostgresqlDatabase supports a `RETURNING` clause on `UPDATE`, `INSERT` and `DELETE` queries. Specifying a `RETURNING` clause allows you to iterate over the rows accessed by the query.

By default, the return values upon execution of the different queries are:

- `INSERT` - auto-incrementing primary key value of the newly-inserted row. When not using an auto-incrementing primary key, Postgres will return the new row's primary key, but SQLite and MySQL will not.
- `UPDATE` - number of rows modified
- `DELETE` - number of rows deleted

When a returning clause is used the return value upon executing a query will be an iterable cursor object.

Postgresql allows, via the `RETURNING` clause, to return data from the rows inserted or modified by a query.

For example, let's say you have an *Update* that deactivates all user accounts whose registration has expired. After deactivating them, you want to send each user an email letting them know their account was deactivated. Rather than writing two queries, a `SELECT` and an `UPDATE`, you can do this in a single `UPDATE` query with a `RETURNING` clause:

```
query = (User
         .update(is_active=False)
         .where(User.registration_expired == True)
```

(continues on next page)

(continued from previous page)

```
         .returning(User))

# Send an email to every user that was deactivated.
for deactivate_user in query.execute():
    send_deactivation_email(deactivated_user.email)
```

The `RETURNING` clause is also available on *`Insert`* and *`Delete`*. When used with `INSERT`, the newly-created rows will be returned. When used with `DELETE`, the deleted rows will be returned.

The only limitation of the `RETURNING` clause is that it can only consist of columns from tables listed in the query's `FROM` clause. To select all columns from a particular table, you can simply pass in the *`Model`* class.

As another example, let's add a user and set their creation-date to the server-generated current timestamp. We'll create and retrieve the new user's ID, Email and the creation timestamp in a single query:

```
query = (User
         .insert(email='foo@bar.com', created=fn.now())
         .returning(User))  # Shorthand for all columns on User.

# When using RETURNING, execute() returns a cursor.
cursor = query.execute()

# Get the user object we just inserted and log the data:
user = cursor[0]
logger.info('Created user %s (id=%s) at %s', user.email, user.id, user.created)
```

By default the cursor will return *`Model`* instances, but you can specify a different row type:

```
data = [{'name': 'charlie'}, {'name': 'huey'}, {'name': 'mickey'}]
query = (User
         .insert_many(data)
         .returning(User.id, User.username)
         .dicts())

for new_user in query.execute():
    print('Added user "%s", id=%s' % (new_user['username'], new_user['id']))
```

Just as with *`Select`* queries, you can specify various *result row types*.

1.8.19 Common Table Expressions

Peewee supports the inclusion of common table expressions (CTEs) in all types of queries. CTEs may be useful for:

- Factoring out a common subquery.
- Grouping or filtering by a column derived in the CTE's result set.
- Writing recursive queries.

To declare a *`Select`* query for use as a CTE, use *`cte()`* method, which wraps the query in a *`CTE`* object. To indicate that a *`CTE`* should be included as part of a query, use the *`Query.with_cte()`* method, passing a list of CTE objects.

Simple Example

For an example, let's say we have some data points that consist of a key and a floating-point value. Let's define our model and populate some test data:

```
class Sample(Model):
    key = TextField()
    value = FloatField()

data = (
    ('a', (1.25, 1.5, 1.75)),
    ('b', (2.1, 2.3, 2.5, 2.7, 2.9)),
    ('c', (3.5, 3.5)))

# Populate data.
for key, values in data:
    Sample.insert_many([(key, value) for value in values],
                       fields=[Sample.key, Sample.value]).execute()
```

Let's use a CTE to calculate, for each distinct key, which values were above-average for that key.

```
# First we'll declare the query that will be used as a CTE. This query
# simply determines the average value for each key.
cte = (Sample
       .select(Sample.key, fn.AVG(Sample.value).alias('avg_value'))
       .group_by(Sample.key)
       .cte('key_avgs', columns=('key', 'avg_value')))

# Now we'll query the sample table, using our CTE to find rows whose value
# exceeds the average for the given key. We'll calculate how far above the
# average the given sample's value is, as well.
query = (Sample
         .select(Sample.key, Sample.value)
         .join(cte, on=(Sample.key == cte.c.key))
         .where(Sample.value > cte.c.avg_value)
         .order_by(Sample.value)
         .with_cte(cte))
```

We can iterate over the samples returned by the query to see which samples had above-average values for their given group:

```
>>> for sample in query:
...     print(sample.key, sample.value)

# 'a', 1.75
# 'b', 2.7
# 'b', 2.9
```

Complex Example

For a more complete example, let's consider the following query which uses multiple CTEs to find per-product sales totals in only the top sales regions. Our model looks like this:

```
class Order(Model):
    region = TextField()
    amount = FloatField()
    product = TextField()
    quantity = IntegerField()
```

Here is how the query might be written in SQL. This example can be found in the postgresql documentation.

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
  ), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales) / 10 FROM regional_sales)
  )
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

With Peewee, we would write:

```
reg_sales = (Order
             .select(Order.region,
                     fn.SUM(Order.amount).alias('total_sales'))
             .group_by(Order.region)
             .cte('regional_sales'))

top_regions = (reg_sales
               .select(reg_sales.c.region)
               .where(reg_sales.c.total_sales > (
                   reg_sales.select(fn.SUM(reg_sales.c.total_sales) / 10)))
               .cte('top_regions'))

query = (Order
         .select(Order.region,
                 Order.product,
                 fn.SUM(Order.quantity).alias('product_units'),
                 fn.SUM(Order.amount).alias('product_sales'))
         .where(Order.region.in_(top_regions.select(top_regions.c.region)))
         .group_by(Order.region, Order.product)
         .with_cte(regional_sales, top_regions))
```

Recursive CTEs

Peewee supports recursive CTEs. Recursive CTEs can be useful when, for example, you have a tree data-structure represented by a parent-link foreign key. Suppose, for example, that we have a hierarchy of categories for an online bookstore. We wish to generate a table showing all categories and their absolute depths, along with the path from the root to the category.

We'll assume the following model definition, in which each category has a foreign-key to its immediate parent category:

```
class Category(Model):
    name = TextField()
    parent = ForeignKeyField('self', backref='children', null=True)
```

To list all categories along with their depth and parents, we can use a recursive CTE:

```
# Define the base case of our recursive CTE. This will be categories that
# have a null parent foreign-key.
Base = Category.alias()
level = Value(1).alias('level')
path = Base.name.alias('path')
base_case = (Base
             .select(Base.name, Base.parent, level, path)
             .where(Base.parent.is_null())
             .cte('base', recursive=True))

# Define the recursive terms.
RTerm = Category.alias()
rlevel = (base_case.c.level + 1).alias('level')
rpath = base_case.c.path.concat('->').concat(RTerm.name).alias('path')
recursive = (RTerm
             .select(RTerm.name, RTerm.parent, rlevel, rpath)
             .join(base_case, on=(RTerm.parent == base_case.c.id)))

# The recursive CTE is created by taking the base case and UNION ALL with
# the recursive term.
cte = base_case.union_all(recursive)

# We will now query from the CTE to get the categories, their levels,  and
# their paths.
query = (cte
         .select_from(cte.c.name, cte.c.level, cte.c.path)
         .order_by(cte.c.path))

# We can now iterate over a list of all categories and print their names,
# absolute levels, and path from root -> category.
for category in query:
    print(category.name, category.level, category.path)

# Example output:
# root, 1, root
# p1, 2, root->p1
# c1-1, 3, root->p1->c1-1
# c1-2, 3, root->p1->c1-2
# p2, 2, root->p2
# c2-1, 3, root->p2->c2-1
```

1.8.20 Foreign Keys and Joins

This section have been moved into its own document: *Relationships and Joins*.

1.9 Query operators

The following types of comparisons are supported by peewee:

Comparison	Meaning
==	x equals y
<	x is less than y
<=	x is less than or equal to y
>	x is greater than y
>=	x is greater than or equal to y
!=	x is not equal to y
<<	x IN y, where y is a list or query
>>	x IS y, where y is None/NULL
%	x LIKE y where y may contain wildcards
**	x ILIKE y where y may contain wildcards
^	x XOR y
~	Unary negation (e.g., NOT x)

Because I ran out of operators to override, there are some additional query operations available as methods:

Method	Meaning
.in_(value)	IN lookup (identical to <<).
.not_in(value)	NOT IN lookup.
.is_null(is_null)	IS NULL or IS NOT NULL. Accepts boolean param.
.contains(substr)	Wild-card search for substring.
.startswith(prefix)	Search for values beginning with prefix.
.endswith(suffix)	Search for values ending with suffix.
.between(low, high)	Search for values between low and high.
.regexp(exp)	Regular expression match (case-sensitive).
.iregexp(exp)	Regular expression match (case-insensitive).
.bin_and(value)	Binary AND.
.bin_or(value)	Binary OR.
.concat(other)	Concatenate two strings or objects using ||.
.distinct()	Mark column for DISTINCT selection.
.collate(collation)	Specify column with the given collation.
.cast(type)	Cast the value of the column to the given type.

To combine clauses using logical operators, use:

Operator	Meaning	Example
&	AND	(User.is_active == True) & (User.is_admin == True)
| (pipe)	OR	(User.is_admin) | (User.is_superuser)
~	NOT (unary negation)	~(User.username.contains('admin'))

Here is how you might use some of these query operators:

```
# Find the user whose username is "charlie".
User.select().where(User.username == 'charlie')

# Find the users whose username is in [charlie, huey, mickey]
User.select().where(User.username.in_(['charlie', 'huey', 'mickey']))

Employee.select().where(Employee.salary.between(50000, 60000))

Employee.select().where(Employee.name.startswith('C'))
```

(continues on next page)

(continued from previous page)

```
Blog.select().where(Blog.title.contains(search_string))
```

Here is how you might combine expressions. Comparisons can be arbitrarily complex.

Note: Note that the actual comparisons are wrapped in parentheses. Python's operator precedence necessitates that comparisons be wrapped in parentheses.

```
# Find any users who are active administrations.
User.select().where(
  (User.is_admin == True) &
  (User.is_active == True))

# Find any users who are either administrators or super-users.
User.select().where(
  (User.is_admin == True) |
  (User.is_superuser == True))

# Find any Tweets by users who are not admins (NOT IN).
admins = User.select().where(User.is_admin == True)
non_admin_tweets = Tweet.select().where(Tweet.user.not_in(admins))

# Find any users who are not my friends (strangers).
friends = User.select().where(User.username.in_(['charlie', 'huey', 'mickey']))
strangers = User.select().where(User.id.not_in(friends))
```

Warning: Although you may be tempted to use python's `in`, `and`, `or` and `not` operators in your query expressions, these **will not work.** The return value of an `in` expression is always coerced to a boolean value. Similarly, `and`, `or` and `not` all treat their arguments as boolean values and cannot be overloaded.

So just remember:

- Use `.in_()` and `.not_in()` instead of `in` and `not in`
- Use `&` instead of `and`
- Use `|` instead of `or`
- Use `~` instead of `not`
- Use `.is_null()` instead of `is None` or `== None`.
- **Don't forget to wrap your comparisons in parentheses when using logical operators.**

For more examples, see the *Expressions* section.

Note: LIKE and ILIKE with SQLite

Because SQLite's `LIKE` operation is case-insensitive by default, peewee will use the SQLite `GLOB` operation for case-sensitive searches. The glob operation uses asterisks for wildcards as opposed to the usual percent-sign. If you are using SQLite and want case-sensitive partial string matching, remember to use asterisks for the wildcard.

1.9.1 Three valued logic

Because of the way SQL handles `NULL`, there are some special operations available for expressing:

- `IS NULL`
- `IS NOT NULL`
- `IN`
- `NOT IN`

While it would be possible to use the `IS NULL` and `IN` operators with the negation operator (`~`), sometimes to get the correct semantics you will need to explicitly use `IS NOT NULL` and `NOT IN`.

The simplest way to use `IS NULL` and `IN` is to use the operator overloads:

```
# Get all User objects whose last login is NULL.
User.select().where(User.last_login >> None)

# Get users whose username is in the given list.
usernames = ['charlie', 'huey', 'mickey']
User.select().where(User.username << usernames)
```

If you don't like operator overloads, you can call the Field methods instead:

```
# Get all User objects whose last login is NULL.
User.select().where(User.last_login.is_null(True))

# Get users whose username is in the given list.
usernames = ['charlie', 'huey', 'mickey']
User.select().where(User.username.in_(usernames))
```

To negate the above queries, you can use unary negation, but for the correct semantics you may need to use the special `IS NOT` and `NOT IN` operators:

```
# Get all User objects whose last login is *NOT* NULL.
User.select().where(User.last_login.is_null(False))

# Using unary negation instead.
User.select().where(~(User.last_login >> None))

# Get users whose username is *NOT* in the given list.
usernames = ['charlie', 'huey', 'mickey']
User.select().where(User.username.not_in(usernames))

# Using unary negation instead.
usernames = ['charlie', 'huey', 'mickey']
User.select().where(~(User.username << usernames))
```

1.9.2 Adding user-defined operators

Because I ran out of python operators to overload, there are some missing operators in peewee, for instance `modulo`. If you find that you need to support an operator that is not in the table above, it is very easy to add your own.

Here is how you might add support for `modulo` in SQLite:

```
from peewee import *
from peewee import Expression # the building block for expressions

def mod(lhs, rhs):
    return Expression(lhs, '%', rhs)
```

Now you can use these custom operators to build richer queries:

```
# Users with even ids.
User.select().where(mod(User.id, 2) == 0)
```

For more examples check out the source to the `playhouse.postgresql_ext` module, as it contains numerous operators specific to postgresql's hstore.

1.9.3 Expressions

Peewee is designed to provide a simple, expressive, and pythonic way of constructing SQL queries. This section will provide a quick overview of some common types of expressions.

There are two primary types of objects that can be composed to create expressions:

- *Field* instances
- SQL aggregations and functions using *fn*

We will assume a simple "User" model with fields for username and other things. It looks like this:

```
class User(Model):
    username = CharField()
    is_admin = BooleanField()
    is_active = BooleanField()
    last_login = DateTimeField()
    login_count = IntegerField()
    failed_logins = IntegerField()
```

Comparisons use the *Query operators*:

```
# username is equal to 'charlie'
User.username == 'charlie'

# user has logged in less than 5 times
User.login_count < 5
```

Comparisons can be combined using **bitwise** *and* and *or*. Operator precedence is controlled by python and comparisons can be nested to an arbitrary depth:

```
# User is both and admin and has logged in today
(User.is_admin == True) & (User.last_login >= today)

# User's username is either charlie or charles
(User.username == 'charlie') | (User.username == 'charles')
```

Comparisons can be used with functions as well:

```
# user's username starts with a 'g' or a 'G':
fn.Lower(fn.Substr(User.username, 1, 1)) == 'g'
```

We can do some fairly interesting things, as expressions can be compared against other expressions. Expressions also support arithmetic operations:

```
# users who entered the incorrect more than half the time and have logged
# in at least 10 times
(User.failed_logins > (User.login_count * .5)) & (User.login_count > 10)
```

Expressions allow us to do *atomic updates*:

```
# when a user logs in we want to increment their login count:
User.update(login_count=User.login_count + 1).where(User.id == user_id)
```

Expressions can be used in all parts of a query, so experiment!

Row values

Many databases support row values, which are similar to Python *tuple* objects. In Peewee, it is possible to use row-values in expressions via `Tuple`. For example,

```
# If for some reason your schema stores dates in separate columns ("year",
# "month" and "day"), you can use row-values to find all rows that happened
# in a given month:
Tuple(Event.year, Event.month) == (2019, 1)
```

The more common use for row-values is to compare against multiple columns from a subquery in a single expression. There are other ways to express these types of queries, but row-values may offer a concise and readable approach.

For example, assume we have a table "EventLog" which contains an event type, an event source, and some metadata. We also have an "IncidentLog", which has incident type, incident source, and metadata columns. We can use row-values to correlate incidents with certain events:

```
class EventLog(Model):
    event_type = TextField()
    source = TextField()
    data = TextField()
    timestamp = TimestampField()

class IncidentLog(Model):
    incident_type = TextField()
    source = TextField()
    traceback = TextField()
    timestamp = TimestampField()

# Get a list of all the incident types and sources that have occured today.
incidents = (IncidentLog
             .select(IncidentLog.incident_type, IncidentLog.source)
             .where(IncidentLog.timestamp >= datetime.date.today()))

# Find all events that correlate with the type and source of the
# incidents that occured today.
events = (EventLog
          .select()
          .where(Tuple(EventLog.event_type, EventLog.source).in_(incidents))
          .order_by(EventLog.timestamp))
```

Other ways to express this type of query would be to use a *join* or to *join on a subquery*. The above example is there just to give you and idea how `Tuple` might be used.

You can also use row-values to update multiple columns in a table, when the new data is derived from a subquery. For an example, see here.

1.9.4 SQL Functions

SQL functions, like `COUNT()` or `SUM()`, can be expressed using the *fn()* helper:

```
# Get all users and the number of tweets they've authored. Sort the
# results from most tweets -> fewest tweets.
query = (User
         .select(User, fn.COUNT(Tweet.id).alias('tweet_count'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User)
         .order_by(fn.COUNT(Tweet.id).desc()))
for user in query:
    print('%s -- %s tweets' % (user.username, user.tweet_count))
```

The `fn` helper exposes any SQL function as if it were a method. The parameters can be fields, values, subqueries, or even nested functions.

Nesting function calls

Suppose you need to want to get a list of all users whose username begins with *a*. There are a couple ways to do this, but one method might be to use some SQL functions like *LOWER* and *SUBSTR*. To use arbitrary SQL functions, use the special *fn()* object to construct queries:

```
# Select the user's id, username and the first letter of their username, lower-cased
first_letter = fn.LOWER(fn.SUBSTR(User.username, 1, 1))
query = User.select(User, first_letter.alias('first_letter'))

# Alternatively we could select only users whose username begins with 'a'
a_users = User.select().where(first_letter == 'a')

>>> for user in a_users:
...     print(user.username)
```

1.9.5 SQL Helper

There are times when you may want to simply pass in some arbitrary sql. You can do this using the special *SQL* class. One use-case is when referencing an alias:

```
# We'll query the user table and annotate it with a count of tweets for
# the given user
query = (User
         .select(User, fn.Count(Tweet.id).alias('ct'))
         .join(Tweet)
         .group_by(User))

# Now we will order by the count, which was aliased to "ct"
query = query.order_by(SQL('ct'))

# You could, of course, also write this as:
query = query.order_by(fn.COUNT(Tweet.id))
```

There are two ways to execute hand-crafted SQL statements with peewee:

1. *Database.execute_sql()* for executing any type of query
2. *RawQuery* for executing SELECT queries and returning model instances.

1.9.6 Security and SQL Injection

By default peewee will parameterize queries, so any parameters passed in by the user will be escaped. The only exception to this rule is if you are writing a raw SQL query or are passing in a SQL object which may contain untrusted data. To mitigate this, ensure that any user-defined data is passed in as a query parameter and not part of the actual SQL query:

```
# Bad! DO NOT DO THIS!
query = MyModel.raw('SELECT * FROM my_table WHERE data = %s' % (user_data,))

# Good. `user_data` will be treated as a parameter to the query.
query = MyModel.raw('SELECT * FROM my_table WHERE data = %s', user_data)

# Bad! DO NOT DO THIS!
query = MyModel.select().where(SQL('Some SQL expression %s' % user_data))

# Good. `user_data` will be treated as a parameter.
query = MyModel.select().where(SQL('Some SQL expression %s', user_data))
```

Note: MySQL and Postgresql use `'%s'` to denote parameters. SQLite, on the other hand, uses `'?'`. Be sure to use the character appropriate to your database. You can also find this parameter by checking `Database.param`.

1.10 Relationships and Joins

In this document we'll cover how Peewee handles relationships between models.

1.10.1 Model definitions

We'll use the following model definitions for our examples:

```
import datetime
from peewee import *


db = SqliteDatabase(':memory:')

class BaseModel(Model):
    class Meta:
        database = db

class User(BaseModel):
    username = TextField()

class Tweet(BaseModel):
    content = TextField()
```

(continues on next page)

(continued from previous page)

```
    timestamp = DateTimeField(default=datetime.datetime.now)
    user = ForeignKeyField(User, backref='tweets')

class Favorite(BaseModel):
    user = ForeignKeyField(User, backref='favorites')
    tweet = ForeignKeyField(Tweet, backref='favorites')
```

Peewee uses *ForeignKeyField* to define foreign-key relationships between models. Every foreign-key field has an implied back-reference, which is exposed as a pre-filtered *Select* query using the provided `backref` attribute.

Creating test data

To follow along with the examples, let's populate this database with some test data:

```
def populate_test_data():
    db.create_tables([User, Tweet, Favorite])

    data = (
        ('huey', ('meow', 'hiss', 'purr')),
        ('mickey', ('woof', 'whine')),
        ('zaizee', ()))
    for username, tweets in data:
        user = User.create(username=username)
        for tweet in tweets:
            Tweet.create(user=user, content=tweet)

    # Populate a few favorites for our users, such that:
    favorite_data = (
        ('huey', ['whine']),
        ('mickey', ['purr']),
        ('zaizee', ['meow', 'purr']))
    for username, favorites in favorite_data:
        user = User.get(User.username == username)
        for content in favorites:
            tweet = Tweet.get(Tweet.content == content)
            Favorite.create(user=user, tweet=tweet)
```

This gives us the following:

User	Tweet	Favorited by
huey	meow	zaizee
huey	hiss	
huey	purr	mickey, zaizee
mickey	woof	
mickey	whine	huey

Attention: In the following examples we will be executing a number of queries. If you are unsure how many queries are being executed, you can add the following code, which will log all queries to the console:

```
import logging
logger = logging.getLogger('peewee')
logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)
```

Note: In SQLite, foreign keys are not enabled by default. Most things, including the Peewee foreign-key API, will work fine, but ON DELETE behaviour will be ignored, even if you explicitly specify `on_delete` in your *ForeignKeyField*. In conjunction with the default *AutoField* behaviour (where deleted record IDs can be reused), this can lead to subtle bugs. To avoid problems, I recommend that you enable foreign-key constraints when using SQLite, by setting `pragmas={'foreign_keys': 1}` when you instantiate *SqliteDatabase*.

```
# Ensure foreign-key constraints are enforced.
db = SqliteDatabase('my_app.db', pragmas={'foreign_keys': 1})
```

1.10.2 Performing simple joins

As an exercise in learning how to perform joins with Peewee, let's write a query to print out all the tweets by "huey". To do this we'll select from the `Tweet` model and join on the `User` model, so we can then filter on the `User.username` field:

```
>>> query = Tweet.select().join(User).where(User.username == 'huey')
>>> for tweet in query:
...     print(tweet.content)
...
meow
hiss
purr
```

Note: We did not have to explicitly specify the join predicate (the "ON" clause), because Peewee inferred from the models that when we joined from Tweet to User, we were joining on the `Tweet.user` foreign-key.

The following code is equivalent, but more explicit:

```
query = (Tweet
         .select()
         .join(User, on=(Tweet.user == User.id))
         .where(User.username == 'huey'))
```

If we already had a reference to the `User` object for "huey", we could use the `User.tweets` back-reference to list all of huey's tweets:

```
>>> huey = User.get(User.username == 'huey')
>>> for tweet in huey.tweets:
...     print(tweet.content)
...
meow
hiss
purr
```

Taking a closer look at `huey.tweets`, we can see that it is just a simple pre-filtered `SELECT` query:

```
>>> huey.tweets
<peewee.ModelSelect at 0x7f0483931fd0>

>>> huey.tweets.sql()
('SELECT "t1"."id", "t1"."content", "t1"."timestamp", "t1"."user_id"
  FROM "tweet" AS "t1" WHERE ("t1"."user_id" = ?)', [1])
```

1.10.3 Joining multiple tables

Let's take another look at joins by querying the list of users and getting the count of how many tweet's they've authored that were favorited. This will require us to join twice: from user to tweet, and from tweet to favorite. We'll add the additional requirement that users should be included who have not created any tweets, as well as users whose tweets have not been favorited. The query, expressed in SQL, would be:

```
SELECT user.username, COUNT(favorite.id)
FROM user
LEFT OUTER JOIN tweet ON tweet.user_id = user.id
LEFT OUTER JOIN favorite ON favorite.tweet_id = tweet.id
GROUP BY user.username
```

Note: In the above query both joins are LEFT OUTER, since a user may not have any tweets or, if they have tweets, none of them may have been favorited.

Peewee has a concept of a *join context*, meaning that whenever we call the `join()` method, we are implicitly joining on the previously-joined model (or if this is the first call, the model we are selecting from). Since we are joining straight through, from user to tweet, then from tweet to favorite, we can simply write:

```
query = (User
         .select(User.username, fn.COUNT(Favorite.id).alias('count'))
         .join(Tweet, JOIN.LEFT_OUTER)  # Joins user -> tweet.
         .join(Favorite, JOIN.LEFT_OUTER)  # Joins tweet -> favorite.
         .group_by(User.username))
```

Iterating over the results:

```
>>> for user in query:
...     print(user.username, user.count)
...
huey 3
mickey 1
zaizee 0
```

For a more complicated example involving multiple joins and switching join contexts, let's find all the tweets by Huey and the number of times they've been favorited. To do this we'll need to perform two joins and we'll also use an aggregate function to calculate the favorite count.

Here is how we would write this query in SQL:

```
SELECT tweet.content, COUNT(favorite.id)
FROM tweet
INNER JOIN user ON tweet.user_id = user.id
LEFT OUTER JOIN favorite ON favorite.tweet_id = tweet.id
WHERE user.username = 'huey'
GROUP BY tweet.content;
```

Note: We use a LEFT OUTER join from tweet to favorite since a tweet may not have any favorites, yet we still wish to display it's content (along with a count of zero) in the result set.

With Peewee, the resulting Python code looks very similar to what we would write in SQL:

```
query = (Tweet
         .select(Tweet.content, fn.COUNT(Favorite.id).alias('count'))
         .join(User)  # Join from tweet -> user.
         .switch(Tweet)  # Move "join context" back to tweet.
         .join(Favorite, JOIN.LEFT_OUTER)  # Join from tweet -> favorite.
         .where(User.username == 'huey')
         .group_by(Tweet.content))
```

Note the call to *switch()* - that instructs Peewee to set the *join context* back to `Tweet`. If we had omitted the explicit call to switch, Peewee would have used `User` (the last model we joined) as the join context and constructed the join from User to Favorite using the `Favorite.user` foreign-key, which would have given us incorrect results.

If we wanted to omit the join-context switching we could instead use the *join_from()* method. The following query is equivalent to the previous one:

```
query = (Tweet
         .select(Tweet.content, fn.COUNT(Favorite.id).alias('count'))
         .join_from(Tweet, User)  # Join tweet -> user.
         .join_from(Tweet, Favorite, JOIN.LEFT_OUTER)  # Join tweet -> favorite.
         .where(User.username == 'huey')
         .group_by(Tweet.content))
```

We can iterate over the results of the above query to print the tweet's content and the favorite count:

```
>>> for tweet in query:
...     print('%s favorited %d times' % (tweet.content, tweet.count))
...
meow favorited 1 times
hiss favorited 0 times
purr favorited 2 times
```

1.10.4 Selecting from multiple sources

If we wished to list all the tweets in the database, along with the username of their author, you might try writing this:

```
>>> for tweet in Tweet.select():
...     print(tweet.user.username, '->', tweet.content)
...
huey -> meow
huey -> hiss
huey -> purr
mickey -> woof
mickey -> whine
```

There is a big problem with the above loop: it executes an additional query for every tweet to look up the `tweet.user` foreign-key. For our small table the performance penalty isn't obvious, but we would find the delays grew as the number of rows increased.

If you're familiar with SQL, you might remember that it's possible to SELECT from multiple tables, allowing us to get the tweet content *and* the username in a single query:

```
SELECT tweet.content, user.username
FROM tweet
INNER JOIN user ON tweet.user_id = user.id;
```

Peewee makes this quite easy. In fact, we only need to modify our query a little bit. We tell Peewee we wish to select `Tweet.content` as well as the `User.username` field, then we include a join from tweet to user. To make it a bit more obvious that it's doing the correct thing, we can ask Peewee to return the rows as dictionaries.

```
>>> for row in Tweet.select(Tweet.content, User.username).join(User).dicts():
...     print(row)
...
{'content': 'meow', 'username': 'huey'}
{'content': 'hiss', 'username': 'huey'}
{'content': 'purr', 'username': 'huey'}
{'content': 'woof', 'username': 'mickey'}
{'content': 'whine', 'username': 'mickey'}
```

Now we'll leave off the call to ".dicts()" and return the rows as `Tweet` objects. Notice that Peewee assigns the `username` value to `tweet.user.username` – NOT `tweet.username`! Because there is a foreign-key from tweet to user, and we have selected fields from both models, Peewee will reconstruct the model-graph for us:

```
>>> for tweet in Tweet.select(Tweet.content, User.username).join(User):
...     print(tweet.user.username, '->', tweet.content)
...
huey -> meow
huey -> hiss
huey -> purr
mickey -> woof
mickey -> whine
```

If we wish to, we can control where Peewee puts the joined `User` instance in the above query, by specifying an `attr` in the `join()` method:

```
>>> query = Tweet.select(Tweet.content, User.username).join(User, attr='author')
>>> for tweet in query:
...     print(tweet.author.username, '->', tweet.content)
...
huey -> meow
huey -> hiss
huey -> purr
mickey -> woof
mickey -> whine
```

Conversely, if we simply wish *all* attributes we select to be attributes of the `Tweet` instance, we can add a call to *`objects()`* at the end of our query (similar to how we called `dicts()`):

```
>>> for tweet in query.objects():
...     print(tweet.username, '->', tweet.content)
...
huey -> meow
(etc)
```

More complex example

As a more complex example, in this query, we will write a single query that selects all the favorites, along with the user who created the favorite, the tweet that was favorited, and that tweet's author.

In SQL we would write:

```
SELECT owner.username, tweet.content, author.username AS author
FROM favorite
INNER JOIN user AS owner ON (favorite.user_id = owner.id)
INNER JOIN tweet ON (favorite.tweet_id = tweet.id)
INNER JOIN user AS author ON (tweet.user_id = author.id);
```

Note that we are selecting from the user table twice - once in the context of the user who created the favorite, and again as the author of the tweet.

With Peewee, we use *Model.alias()* to alias a model class so it can be referenced twice in a single query:

```
Owner = User.alias()
query = (Favorite
         .select(Favorite, Tweet.content, User.username, Owner.username)
         .join(Owner)  # Join favorite -> user (owner of favorite).
         .switch(Favorite)
         .join(Tweet)  # Join favorite -> tweet
         .join(User))   # Join tweet -> user
```

We can iterate over the results and access the joined values in the following way. Note how Peewee has resolved the fields from the various models we selected and reconstructed the model graph:

```
>>> for fav in query:
...     print(fav.user.username, 'liked', fav.tweet.content, 'by', fav.tweet.user.
↪username)
...
huey liked whine by mickey
mickey liked purr by huey
zaizee liked meow by huey
zaizee liked purr by huey
```

1.10.5 Subqueries

Peewee allows you to join on any table-like object, including subqueries or common table expressions (CTEs). To demonstrate joining on a subquery, let's query for all users and their latest tweet.

Here is the SQL:

```
SELECT tweet.*, user.*
FROM tweet
INNER JOIN (
    SELECT latest.user_id, MAX(latest.timestamp) AS max_ts
    FROM tweet AS latest
    GROUP BY latest.user_id) AS latest_query
ON ((tweet.user_id = latest_query.user_id) AND (tweet.timestamp = latest_query.max_
↪ts))
INNER JOIN user ON (tweet.user_id = user.id)
```

We'll do this by creating a subquery which selects each user and the timestamp of their latest tweet. Then we can query the tweets table in the outer query and join on the user and timestamp combination from the subquery.

```
# Define our subquery first. We'll use an alias of the Tweet model, since
# we will be querying from the Tweet model directly in the outer query.
Latest = Tweet.alias()
latest_query = (Latest
                .select(Latest.user, fn.MAX(Latest.timestamp).alias('max_ts'))
```

(continues on next page)

(continued from previous page)

```
                .group_by(Latest.user)
                .alias('latest_query'))

# Our join predicate will ensure that we match tweets based on their
# timestamp *and* user_id.
predicate = ((Tweet.user == latest_query.c.user_id) &
             (Tweet.timestamp == latest_query.c.max_ts))

# We put it all together, querying from tweet and joining on the subquery
# using the above predicate.
query = (Tweet
         .select(Tweet, User)  # Select all columns from tweet and user.
         .join(latest_query, on=predicate)  # Join tweet -> subquery.
         .join_from(Tweet, User))  # Join from tweet -> user.
```

Iterating over the query, we can see each user and their latest tweet.

```
>>> for tweet in query:
...     print(tweet.user.username, '->', tweet.content)
...
huey -> purr
mickey -> whine
```

There are a couple things you may not have seen before in the code we used to create the query in this section:

- We used *join_from()* to explicitly specify the join context. We wrote `.join_from(Tweet, User)`, which is equivalent to `.switch(Tweet).join(User)`.
- We referenced columns in the subquery using the magic `.c` attribute, for example `latest_query.c.max_ts`. The `.c` attribute is used to dynamically create column references.
- Instead of passing individual fields to `Tweet.select()`, we passed the `Tweet` and `User` models. This is shorthand for selecting all fields on the given model.

Common-table Expressions

In the previous section we joined on a subquery, but we could just as easily have used a *common-table expression (CTE)*. We will repeat the same query as before, listing users and their latest tweets, but this time we will do it using a CTE.

Here is the SQL:

```
WITH latest AS (
    SELECT user_id, MAX(timestamp) AS max_ts
    FROM tweet
    GROUP BY user_id)
SELECT tweet.*, user.*
FROM tweet
INNER JOIN latest
    ON ((latest.user_id = tweet.user_id) AND (latest.max_ts = tweet.timestamp))
INNER JOIN user
    ON (tweet.user_id = user.id)
```

This example looks very similar to the previous example with the subquery:

```
# Define our CTE first. We'll use an alias of the Tweet model, since
# we will be querying from the Tweet model directly in the main query.
Latest = Tweet.alias()
cte = (Latest
       .select(Latest.user, fn.MAX(Latest.timestamp).alias('max_ts'))
       .group_by(Latest.user)
       .cte('latest'))

# Our join predicate will ensure that we match tweets based on their
# timestamp *and* user_id.
predicate = ((Tweet.user == cte.c.user_id) &
             (Tweet.timestamp == cte.c.max_ts))

# We put it all together, querying from tweet and joining on the CTE
# using the above predicate.
query = (Tweet
         .select(Tweet, User)  # Select all columns from tweet and user.
         .join(cte, on=predicate)  # Join tweet -> CTE.
         .join_from(Tweet, User)  # Join from tweet -> user.
         .with_cte(cte))
```

We can iterate over the result-set, which consists of the latest tweets for each user:

```
>>> for tweet in query:
...     print(tweet.user.username, '->', tweet.content)
...
huey -> purr
mickey -> whine
```

Note: For more information about using CTEs, including information on writing recursive CTEs, see the *Common Table Expressions* section of the "Querying" document.

1.10.6 Multiple foreign-keys to the same Model

When there are multiple foreign keys to the same model, it is good practice to explicitly specify which field you are joining on.

Referring back to the *example app's models*, consider the *Relationship* model, which is used to denote when one user follows another. Here is the model definition:

```
class Relationship(BaseModel):
    from_user = ForeignKeyField(User, backref='relationships')
    to_user = ForeignKeyField(User, backref='related_to')

    class Meta:
        indexes = (
            # Specify a unique multi-column index on from/to-user.
            (('from_user', 'to_user'), True),
        )
```

Since there are two foreign keys to *User*, we should always specify which field we are using in a join.

For example, to determine which users I am following, I would write:

```
(User
 .select()
 .join(Relationship, on=Relationship.to_user)
 .where(Relationship.from_user == charlie))
```

On the other hand, if I wanted to determine which users are following me, I would instead join on the *from_user* column and filter on the relationship's *to_user*:

```
(User
 .select()
 .join(Relationship, on=Relationship.from_user)
 .where(Relationship.to_user == charlie))
```

1.10.7 Joining on arbitrary fields

If a foreign key does not exist between two tables you can still perform a join, but you must manually specify the join predicate.

In the following example, there is no explicit foreign-key between *User* and *ActivityLog*, but there is an implied relationship between the *ActivityLog.object_id* field and *User.id*. Rather than joining on a specific *Field*, we will join using an *Expression*.

```
user_log = (User
            .select(User, ActivityLog)
            .join(ActivityLog, on=(User.id == ActivityLog.object_id), attr='log')
            .where(
                (ActivityLog.activity_type == 'user_activity') &
                (User.username == 'charlie')))

for user in user_log:
    print(user.username, user.log.description)

#### Print something like ####
charlie logged in
charlie posted a tweet
charlie retweeted
charlie posted a tweet
charlie logged out
```

Note: Recall that we can control the attribute Peewee will assign the joined instance to by specifying the `attr` parameter in the `join()` method. In the previous example, we used the following *join*:

```
join(ActivityLog, on=(User.id == ActivityLog.object_id), attr='log')
```

Then when iterating over the query, we were able to directly access the joined *ActivityLog* without incurring an additional query:

```
for user in user_log:
    print(user.username, user.log.description)
```

1.10.8 Self-joins

Peewee supports constructing queries containing a self-join.

Using model aliases

To join on the same model (table) twice, it is necessary to create a model alias to represent the second instance of the table in a query. Consider the following model:

```
class Category(Model):
    name = CharField()
    parent = ForeignKeyField('self', backref='children')
```

What if we wanted to query all categories whose parent category is *Electronics*. One way would be to perform a self-join:

```
Parent = Category.alias()
query = (Category
         .select()
         .join(Parent, on=(Category.parent == Parent.id))
         .where(Parent.name == 'Electronics'))
```

When performing a join that uses a *ModelAlias*, it is necessary to specify the join condition using the `on` keyword argument. In this case we are joining the category with its parent category.

Using subqueries

Another less common approach involves the use of subqueries. Here is another way we might construct a query to get all the categories whose parent category is *Electronics* using a subquery:

```
Parent = Category.alias()
join_query = Parent.select().where(Parent.name == 'Electronics')

# Subqueries used as JOINs need to have an alias.
join_query = join_query.alias('jq')

query = (Category
         .select()
         .join(join_query, on=(Category.parent == join_query.c.id)))
```

This will generate the following SQL query:

```
SELECT t1."id", t1."name", t1."parent_id"
FROM "category" AS t1
INNER JOIN (
  SELECT t2."id"
  FROM "category" AS t2
  WHERE (t2."name" = ?)) AS jq ON (t1."parent_id" = "jq"."id")
```

To access the `id` value from the subquery, we use the `.c` magic lookup which will generate the appropriate SQL expression:

```
Category.parent == join_query.c.id
# Becomes: (t1."parent_id" = "jq"."id")
```

1.10.9 Implementing Many to Many

Peewee provides a field for representing many-to-many relationships, much like Django does. This feature was added due to many requests from users, but I strongly advocate against using it, since it conflates the idea of a field with a

junction table and hidden joins. It's just a nasty hack to provide convenient accessors.

To implement many-to-many **correctly** with peewee, you will therefore create the intermediary table yourself and query through it:

```
class Student(Model):
    name = CharField()

class Course(Model):
    name = CharField()

class StudentCourse(Model):
    student = ForeignKeyField(Student)
    course = ForeignKeyField(Course)
```

To query, let's say we want to find students who are enrolled in math class:

```
query = (Student
         .select()
         .join(StudentCourse)
         .join(Course)
         .where(Course.name == 'math'))
for student in query:
    print(student.name)
```

To query what classes a given student is enrolled in:

```
courses = (Course
           .select()
           .join(StudentCourse)
           .join(Student)
           .where(Student.name == 'da vinci'))

for course in courses:
    print(course.name)
```

To efficiently iterate over a many-to-many relation, i.e., list all students and their respective courses, we will query the *through* model `StudentCourse` and *precompute* the Student and Course:

```
query = (StudentCourse
         .select(StudentCourse, Student, Course)
         .join(Course)
         .switch(StudentCourse)
         .join(Student)
         .order_by(Student.name))
```

To print a list of students and their courses you might do the following:

```
for student_course in query:
    print(student_course.student.name, '->', student_course.course.name)
```

Since we selected all fields from `Student` and `Course` in the *select* clause of the query, these foreign key traversals are "free" and we've done the whole iteration with just 1 query.

ManyToManyField

The *ManyToManyField* provides a *field-like* API over many-to-many fields. For all but the simplest many-to-many situations, you're better off using the standard peewee APIs. But, if your models are very simple and your querying

needs are not very complex, *ManyToManyField* may work.

Modeling students and courses using *ManyToManyField*:

```
from peewee import *

db = SqliteDatabase('school.db')

class BaseModel(Model):
    class Meta:
        database = db

class Student(BaseModel):
    name = CharField()

class Course(BaseModel):
    name = CharField()
    students = ManyToManyField(Student, backref='courses')

StudentCourse = Course.students.get_through_model()

db.create_tables([
    Student,
    Course,
    StudentCourse])

# Get all classes that "huey" is enrolled in:
huey = Student.get(Student.name == 'Huey')
for course in huey.courses.order_by(Course.name):
    print(course.name)

# Get all students in "English 101":
engl_101 = Course.get(Course.name == 'English 101')
for student in engl_101.students:
    print(student.name)

# When adding objects to a many-to-many relationship, we can pass
# in either a single model instance, a list of models, or even a
# query of models:
huey.courses.add(Course.select().where(Course.name.contains('English')))

engl_101.students.add(Student.get(Student.name == 'Mickey'))
engl_101.students.add([
    Student.get(Student.name == 'Charlie'),
    Student.get(Student.name == 'Zaizee')])

# The same rules apply for removing items from a many-to-many:
huey.courses.remove(Course.select().where(Course.name.startswith('CS')))

engl_101.students.remove(huey)

# Calling .clear() will remove all associated objects:
cs_150.students.clear()
```

Attention: Before many-to-many relationships can be added, the objects being referenced will need to be saved first. In order to create relationships in the many-to-many through table, Peewee needs to know the primary keys of the models being referenced.

Warning: It is **strongly recommended** that you do not attempt to subclass models containing `ManyToManyField` instances.

A `ManyToManyField`, despite its name, is not a field in the usual sense. Instead of being a column on a table, the many-to-many field covers the fact that behind-the-scenes there's actually a separate table with two foreign-key pointers (the *through table*).

Therefore, when a subclass is created that inherits a many-to-many field, what actually needs to be inherited is the *through table*. Because of the potential for subtle bugs, Peewee does not attempt to automatically subclass the through model and modify its foreign-key pointers. As a result, many-to-many fields typically will not work with inheritance.

For more examples, see:

- `ManyToManyField.add()`
- `ManyToManyField.remove()`
- `ManyToManyField.clear()`
- `ManyToManyField.get_through_model()`

1.10.10 Avoiding the N+1 problem

The *N+1 problem* refers to a situation where an application performs a query, then for each row of the result set, the application performs at least one other query (another way to conceptualize this is as a nested loop). In many cases, these *n* queries can be avoided through the use of a SQL join or subquery. The database itself may do a nested loop, but it will usually be more performant than doing *n* queries in your application code, which involves latency communicating with the database and may not take advantage of indices or other optimizations employed by the database when joining or executing a subquery.

Peewee provides several APIs for mitigating *N+1* query behavior. Recollecting the models used throughout this document, *User* and *Tweet*, this section will try to outline some common *N+1* scenarios, and how peewee can help you avoid them.

Attention: In some cases, N+1 queries will not result in a significant or measurable performance hit. It all depends on the data you are querying, the database you are using, and the latency involved in executing queries and retrieving results. As always when making optimizations, profile before and after to ensure the changes do what you expect them to.

List recent tweets

The twitter timeline displays a list of tweets from multiple users. In addition to the tweet's content, the username of the tweet's author is also displayed. The N+1 scenario here would be:

1. Fetch the 10 most recent tweets.
2. For each tweet, select the author (10 queries).

By selecting both tables and using a *join*, peewee makes it possible to accomplish this in a single query:

```
query = (Tweet
         .select(Tweet, User)  # Note that we are selecting both models.
         .join(User)  # Use an INNER join because every tweet has an author.
```

(continues on next page)

(continued from previous page)

```
         .order_by(Tweet.id.desc())  # Get the most recent tweets.
         .limit(10))

for tweet in query:
    print(tweet.user.username, '-', tweet.message)
```

Without the join, accessing `tweet.user.username` would trigger a query to resolve the foreign key `tweet.user` and retrieve the associated user. But since we have selected and joined on `User`, peewee will automatically resolve the foreign-key for us.

Note: This technique is discussed in more detail in *Selecting from multiple sources*.

List users and all their tweets

Let's say you want to build a page that shows several users and all of their tweets. The N+1 scenario would be:

1. Fetch some users.
2. For each user, fetch their tweets.

This situation is similar to the previous example, but there is one important difference: when we selected tweets, they only have a single associated user, so we could directly assign the foreign key. The reverse is not true, however, as one user may have any number of tweets (or none at all).

Peewee provides an approach to avoiding *O(n)* queries in this situation. Fetch users first, then fetch all the tweets associated with those users. Once peewee has the big list of tweets, it will assign them out, matching them with the appropriate user. This method is usually faster but will involve a query for each table being selected.

Using prefetch

peewee supports pre-fetching related data using sub-queries. This method requires the use of a special API, *prefetch()*. Prefetch, as its name implies, will eagerly load the appropriate tweets for the given users using subqueries. This means instead of *O(n)* queries for *n* rows, we will do *O(k)* queries for *k* tables.

Here is an example of how we might fetch several users and any tweets they created within the past week.

```
week_ago = datetime.date.today() - datetime.timedelta(days=7)
users = User.select()
tweets = (Tweet
          .select()
          .where(Tweet.timestamp >= week_ago))

# This will perform two queries.
users_with_tweets = prefetch(users, tweets)

for user in users_with_tweets:
    print(user.username)
    for tweet in user.tweets:
        print('  ', tweet.message)
```

Note: Note that neither the `User` query, nor the `Tweet` query contained a JOIN clause. When using *prefetch()* you do not need to specify the join.

prefetch() can be used to query an arbitrary number of tables. Check the API documentation for more examples.

Some things to consider when using *prefetch()*:

- Foreign keys must exist between the models being prefetched.
- *LIMIT* works as you'd expect on the outer-most query, but may be difficult to implement correctly if trying to limit the size of the sub-selects.

1.11 API Documentation

This document specifies Peewee's APIs.

1.11.1 Database

class Database(*database*[, *thread_safe=True*[, *autorollback=False*[, *field_types=None*[, *operations=None*[, *autoconnect=True*[, ***kwargs*]]]]]])

Parameters

- **database** (*str*) – Database name or filename for SQLite (or `None` to *defer initialization*, in which case you must call *Database.init()*, specifying the database name).
- **thread_safe** (*bool*) – Whether to store connection state in a thread-local.
- **autorollback** (*bool*) – Automatically rollback queries that fail when **not** in an explicit transaction.
- **field_types** (*dict*) – A mapping of additional field types to support.
- **operations** (*dict*) – A mapping of additional operations to support.
- **autoconnect** (*bool*) – Automatically connect to database if attempting to execute a query on a closed database.
- **kwargs** – Arbitrary keyword arguments that will be passed to the database driver when a connection is created, for example `password`, `host`, etc.

The *Database* is responsible for:

- Executing queries
- Managing connections
- Transactions
- Introspection

Note: The database can be instantiated with `None` as the database name if the database is not known until run-time. In this way you can create a database instance and then configure it elsewhere when the settings are known. This is called *deferred* initialization*.

Examples:

```
# Sqlite database using WAL-mode and 32MB page-cache.
db = SqliteDatabase('app.db', pragmas={
    'journal_mode': 'wal',
    'cache_size': -32 * 1000})
```

(continues on next page)

(continued from previous page)

```
# Postgresql database on remote host.
db = PostgresqlDatabase('my_app', user='postgres', host='10.1.0.3',
                        password='secret')
```

Deferred initialization example:

```
db = PostgresqlDatabase(None)

class BaseModel(Model):
    class Meta:
        database = db

# Read database connection info from env, for example:
db_name = os.environ['DATABASE']
db_host = os.environ['PGHOST']

# Initialize database.
db.init(db_name, host=db_host, user='postgres')
```

param = '?'
String used as parameter placeholder in SQL queries.

quote = '"'
Type of quotation-mark to use to denote entities such as tables or columns.

init(*database*[, ***kwargs*])

Parameters

- **database** (*str*) – Database name or filename for SQLite.
- **kwargs** – Arbitrary keyword arguments that will be passed to the database driver when a connection is created, for example `password`, `host`, etc.

Initialize a *deferred* database. See *Run-time database configuration* for more info.

__enter__()
The *Database* instance can be used as a context-manager, in which case a connection will be held open for the duration of the wrapped block.

Additionally, any SQL executed within the wrapped block will be executed in a transaction.

connection_context()
Create a context-manager that will hold open a connection for the duration of the wrapped block.

Example:

```
def on_app_startup():
    # When app starts up, create the database tables, being sure
    # the connection is closed upon completion.
    with database.connection_context():
        database.create_tables(APP_MODELS)
```

connect([*reuse_if_open=False*])

Parameters **reuse_if_open** (*bool*) – Do not raise an exception if a connection is already opened.

Returns whether a new connection was opened.

Return type bool

Raises `OperationalError` if connection already open and `reuse_if_open` is not set to `True`.

Open a connection to the database.

close()

Returns Whether a connection was closed. If the database was already closed, this returns `False`.

Return type bool

Close the connection to the database.

is_closed()

Returns return `True` if database is closed, `False` if open.

Return type bool

connection()

Return the open connection. If a connection is not open, one will be opened. The connection will be whatever the underlying database-driver uses to encapsulate a database connection.

cursor(*commit=None*)

Parameters **commit** – For internal use.

Return a `cursor` object on the current connection. If a connection is not open, one will be opened. The cursor will be whatever the underlying database-driver uses to encapsulate a database cursor.

execute_sql(*sql*, *params=None*, *commit=SENTINEL*)

Parameters

- **sql** (*str*) – SQL string to execute.
- **params** (*tuple*) – Parameters for query.
- **commit** – Boolean flag to override the default commit logic.

Returns cursor object.

Execute a SQL query and return a cursor over the results.

execute(*query*, *commit=SENTINEL*, ***context_options*)

Parameters

- **query** – A *Query* instance.
- **commit** – Boolean flag to override the default commit logic.
- **context_options** – Arbitrary options to pass to the SQL generator.

Returns cursor object.

Execute a SQL query by compiling a `Query` instance and executing the resulting SQL.

last_insert_id(*cursor*, *query_type=None*)

Parameters **cursor** – cursor object.

Returns primary key of last-inserted row.

rows_affected(*cursor*)

Parameters **cursor** – cursor object.

Returns number of rows modified by query.

in_transaction()

Returns whether or not a transaction is currently open.

Return type bool

atomic()

Create a context-manager which runs any queries in the wrapped block in a transaction (or save-point if blocks are nested).

Calls to *atomic()* can be nested.

atomic() can also be used as a decorator.

Example code:

```
with db.atomic() as txn:
    perform_operation()

    with db.atomic() as nested_txn:
        perform_another_operation()
```

Transactions and save-points can be explicitly committed or rolled-back within the wrapped block. If this occurs, a new transaction or savepoint is begun after the commit/rollback.

Example:

```
with db.atomic() as txn:
    User.create(username='mickey')
    txn.commit()  # Changes are saved and a new transaction begins.

    User.create(username='huey')
    txn.rollback()  # "huey" will not be saved.

    User.create(username='zaizee')

# Print the usernames of all users.
print [u.username for u in User.select()]

# Prints ["mickey", "zaizee"]
```

manual_commit()

Create a context-manager which disables all transaction management for the duration of the wrapped block.

Example:

```
with db.manual_commit():
    db.begin()  # Begin transaction explicitly.
    try:
        user.delete_instance(recursive=True)
    except:
        db.rollback()  # Rollback -- an error occurred.
        raise
    else:
        try:
            db.commit()  # Attempt to commit changes.
        except:
```

(continues on next page)

(continued from previous page)

```
            db.rollback()  # Error committing, rollback.
            raise
```

The above code is equivalent to the following:

```
with db.atomic():
    user.delete_instance(recursive=True)
```

session_start()
Begin a new transaction (without using a context-manager or decorator). This method is useful if you intend to execute a sequence of operations inside a transaction, but using a decorator or context-manager would not be appropriate.

Note: It is strongly advised that you use the *Database.atomic()* method whenever possible for managing transactions/savepoints. The `atomic` method correctly manages nesting, uses the appropriate construction (e.g., transaction-vs-savepoint), and always cleans up after itself.

The *session_start()* method should only be used if the sequence of operations does not easily lend itself to wrapping using either a context-manager or decorator.

Warning: You must *always* call either *session_commit()* or *session_rollback()* after calling the `session_start` method.

session_commit()
Commit any changes made during a transaction begun with *session_start()*.

session_rollback()
Roll back any changes made during a transaction begun with *session_start()*.

transaction()
Create a context-manager that runs all queries in the wrapped block in a transaction.

Warning: Calls to `transaction` cannot be nested. Only the top-most call will take effect. Rolling-back or committing a nested transaction context-manager has undefined behavior.

savepoint()
Create a context-manager that runs all queries in the wrapped block in a savepoint. Savepoints can be nested arbitrarily.

Warning: Calls to `savepoint` must occur inside of a transaction.

begin()
Begin a transaction when using manual-commit mode.

Note: This method should only be used in conjunction with the *manual_commit()* context manager.

---

commit()
Manually commit the currently-active transaction.

Note: This method should only be used in conjunction with the *manual_commit()* context manager.

rollback()
Manually roll-back the currently-active transaction.

Note: This method should only be used in conjunction with the *manual_commit()* context manager.

batch_commit(*it*, *n*)

Parameters

- **it** (*iterable*) – an iterable whose items will be yielded.
- **n** (*int*) – commit every *n* items.

Returns an equivalent iterable to the one provided, with the addition that groups of *n* items will be yielded in a transaction.

The purpose of this method is to simplify batching large operations, such as inserts, updates, etc. You pass in an iterable and the number of items-per-batch, and the items will be returned by an equivalent iterator that wraps each batch in a transaction.

Example:

```
# Some list or iterable containing data to insert.
row_data = [{'username': 'u1'}, {'username': 'u2'}, ...]

# Insert all data, committing every 100 rows. If, for example,
# there are 789 items in the list, then there will be a total of
# 8 transactions (7x100 and 1x89).
for row in db.batch_commit(row_data, 100):
    User.create(**row)
```

An alternative that may be more efficient is to batch the data into a multi-value `INSERT` statement (for example, using *Model.insert_many()*):

```
with db.atomic():
    for idx in range(0, len(row_data), 100):
        # Insert 100 rows at a time.
        rows = row_data[idx:idx + 100]
        User.insert_many(rows).execute()
```

table_exists(*table*[, *schema=None*])

Parameters

- **table** (*str*) – Table name.
- **schema** (*str*) – Schema name (optional).

Returns `bool` indicating whether table exists.

get_tables([*schema=None*])

Parameters **schema** (*str*) – Schema name (optional).

Returns a list of table names in the database.

get_indexes(*table*[, *schema=None*])

Parameters

- **table** (*str*) – Table name.
- **schema** (*str*) – Schema name (optional).

Return a list of `IndexMetadata` tuples.

Example:

```
print(db.get_indexes('entry'))
[IndexMetadata(
     name='entry_public_list',
     sql='CREATE INDEX "entry_public_list" ...',
     columns=['timestamp'],
     unique=False,
     table='entry'),
 IndexMetadata(
     name='entry_slug',
     sql='CREATE UNIQUE INDEX "entry_slug" ON "entry" ("slug")',
     columns=['slug'],
     unique=True,
     table='entry')]
```

get_columns(*table*[, *schema=None*])

Parameters

- **table** (*str*) – Table name.
- **schema** (*str*) – Schema name (optional).

Return a list of `ColumnMetadata` tuples.

Example:

```
print(db.get_columns('entry'))
[ColumnMetadata(
     name='id',
     data_type='INTEGER',
     null=False,
     primary_key=True,
     table='entry'),
 ColumnMetadata(
     name='title',
     data_type='TEXT',
     null=False,
     primary_key=False,
     table='entry'),
 ...]
```

get_primary_keys(*table*[, *schema=None*])

Parameters

- **table** (*str*) – Table name.
- **schema** (*str*) – Schema name (optional).

Return a list of column names that comprise the primary key.

Example:

```
print(db.get_primary_keys('entry'))
['id']
```

get_foreign_keys(*table*[, *schema=None*])

Parameters

- **table** (*str*) – Table name.
- **schema** (*str*) – Schema name (optional).

Return a list of `ForeignKeyMetadata` tuples for keys present on the table.

Example:

```
print(db.get_foreign_keys('entrytag'))
[ForeignKeyMetadata(
     column='entry_id',
     dest_table='entry',
     dest_column='id',
     table='entrytag'),
 ...]
```

get_views([*schema=None*])

Parameters **schema** (*str*) – Schema name (optional).

Return a list of `ViewMetadata` tuples for VIEWs present in the database.

Example:

```
print(db.get_views())
[ViewMetadata(
     name='entries_public',
     sql='CREATE VIEW entries_public AS SELECT ... '),
 ...]
```

sequence_exists(*seq*)

Parameters **seq** (*str*) – Name of sequence.

Returns Whether sequence exists.

Return type bool

create_tables(*models*[, ***options*])

Parameters

- **models** (*list*) – A list of *Model* classes.
- **options** – Options to specify when calling *Model.create_table()*.

Create tables, indexes and associated metadata for the given list of models.

Dependencies are resolved so that tables are created in the appropriate order.

drop_tables(*models*[, ***options*])

Parameters

- **models** (*list*) – A list of *Model* classes.
- **kwargs** – Options to specify when calling *Model.drop_table()*.

Drop tables, indexes and associated metadata for the given list of models.

Dependencies are resolved so that tables are dropped in the appropriate order.

bind(*models*[, *bind_refs=True*[, *bind_backrefs=True*]])

Parameters

- **models** (*list*) – One or more *Model* classes to bind.
- **bind_refs** (*bool*) – Bind related models.
- **bind_backrefs** (*bool*) – Bind back-reference related models.

Bind the given list of models, and specified relations, to the database.

bind_ctx(*models*[, *bind_refs=True*[, *bind_backrefs=True*]])

Parameters

- **models** (*list*) – List of models to bind to the database.
- **bind_refs** (*bool*) – Bind models that are referenced using foreign-keys.
- **bind_backrefs** (*bool*) – Bind models that reference the given model with a foreign-key.

Create a context-manager that binds (associates) the given models with the current database for the duration of the wrapped block.

Example:

```
MODELS = (User, Account, Note)

# Bind the given models to the db for the duration of wrapped block.
def use_test_database(fn):
    @wraps(fn)
    def inner(self):
        with test_db.bind_ctx(MODELS):
            test_db.create_tables(MODELS)
            try:
                fn(self)
            finally:
                test_db.drop_tables(MODELS)
    return inner


class TestSomething(TestCase):
    @use_test_database
    def test_something(self):
        # ... models are bound to test database ...
        pass
```

extract_date(*date_part*, *date_field*)

Parameters

- **date_part** (*str*) – date part to extract, e.g. 'year'.
- **date_field** (Node) – a SQL node containing a date/time, for example a *DateTimeField*.

Returns a SQL node representing a function call that will return the provided date part.

Provides a compatible interface for extracting a portion of a datetime.

truncate_date(*date_part*, *date_field*)

Parameters

- **date_part** (*str*) – date part to truncate to, e.g. 'day'.
- **date_field** (Node) – a SQL node containing a date/time, for example a *DateTimeField*.

Returns a SQL node representing a function call that will return the truncated date part.

Provides a compatible interface for truncating a datetime to the given resolution.

random()

Returns a SQL node representing a function call that returns a random value.

A compatible interface for calling the appropriate random number generation function provided by the database. For Postgres and Sqlite, this is equivalent to `fn.random()`, for MySQL `fn.rand()`.

class SqliteDatabase(*database*[, *pragmas=None*[, *timeout=5*[, ***kwargs*]]])

Parameters

- **pragmas** – Either a dictionary or a list of 2-tuples containing pragma key and value to set every time a connection is opened.
- **timeout** – Set the busy-timeout on the SQLite driver (in seconds).

Sqlite database implementation. *SqliteDatabase* that provides some advanced features only offered by Sqlite.

- Register custom aggregates, collations and functions
- Load C extensions
- Advanced transactions (specify lock type)
- For even more features, see *SqliteExtDatabase*.

Example of initializing a database and configuring some PRAGMAs:

```
db = SqliteDatabase('my_app.db', pragmas=(
    ('cache_size', -16000),  # 16MB
    ('journal_mode', 'wal'),  # Use write-ahead-log journal mode.
))

# Alternatively, pragmas can be specified using a dictionary.
db = SqliteDatabase('my_app.db', pragmas={'journal_mode': 'wal'})
```

pragma(*key*[, *value=SENTINEL*[, *permanent=False*]])

Parameters

- **key** – Setting name.
- **value** – New value for the setting (optional).
- **permanent** – Apply this pragma whenever a connection is opened.

Execute a PRAGMA query once on the active connection. If a value is not specified, then the current value will be returned.

If `permanent` is specified, then the PRAGMA query will also be executed whenever a new connection is opened, ensuring it is always in-effect.

Note: By default this only affects the current connection. If the PRAGMA being executed is not persistent, then you must specify `permanent=True` to ensure the pragma is set on subsequent connections.

cache_size
Get or set the cache_size pragma for the current connection.

foreign_keys
Get or set the foreign_keys pragma for the current connection.

journal_mode
Get or set the journal_mode pragma.

journal_size_limit
Get or set the journal_size_limit pragma.

mmap_size
Get or set the mmap_size pragma for the current connection.

page_size
Get or set the page_size pragma.

read_uncommitted
Get or set the read_uncommitted pragma for the current connection.

synchronous
Get or set the synchronous pragma for the current connection.

wal_autocheckpoint
Get or set the wal_autocheckpoint pragma for the current connection.

timeout
Get or set the busy timeout (seconds).

register_aggregate(*klass*[, *name=None*[, *num_params=-1*]])

Parameters

- **klass** – Class implementing aggregate API.
- **name** (*str*) – Aggregate function name (defaults to name of class).
- **num_params** (*int*) – Number of parameters the aggregate accepts, or -1 for any number.

Register a user-defined aggregate function.

The function will be registered each time a new connection is opened. Additionally, if a connection is already open, the aggregate will be registered with the open connection.

aggregate([*name=None*[, *num_params=-1*]])

Parameters

- **name** (*str*) – Name of the aggregate (defaults to class name).
- **num_params** (*int*) – Number of parameters the aggregate accepts, or -1 for any number.

Class decorator to register a user-defined aggregate function.

Example:

```
@db.aggregate('md5')
class MD5(object):
    def initialize(self):
```

(continues on next page)

(continued from previous page)

```
        self.md5 = hashlib.md5()

    def step(self, value):
        self.md5.update(value)

    def finalize(self):
        return self.md5.hexdigest()


@db.aggregate()
class Product(object):
    '''Like SUM() except calculates cumulative product.'''
    def __init__(self):
        self.product = 1

    def step(self, value):
        self.product *= value

    def finalize(self):
        return self.product
```

register_collation(*fn*[, *name=None*])

Parameters

- **fn** – The collation function.
- **name** (*str*) – Name of collation (defaults to function name)

Register a user-defined collation. The collation will be registered each time a new connection is opened. Additionally, if a connection is already open, the collation will be registered with the open connection.

collation([*name=None*])

Parameters **name** (*str*) – Name of collation (defaults to function name)

Decorator to register a user-defined collation.

Example:

```
@db.collation('reverse')
def collate_reverse(s1, s2):
    return -cmp(s1, s2)

# Usage:
Book.select().order_by(collate_reverse.collation(Book.title))

# Equivalent:
Book.select().order_by(Book.title.asc(collation='reverse'))
```

As you might have noticed, the original `collate_reverse` function has a special attribute called `collation` attached to it. This extra attribute provides a shorthand way to generate the SQL necessary to use our custom collation.

register_function(*fn*[, *name=None*[, *num_params=-1*]])

Parameters

- **fn** – The user-defined scalar function.
- **name** (*str*) – Name of function (defaults to function name)

- **num_params** (*int*) – Number of arguments the function accepts, or -1 for any number.

Register a user-defined scalar function. The function will be registered each time a new connection is opened. Additionally, if a connection is already open, the function will be registered with the open connection.

func([*name=None*[, *num_params=-1*]])

Parameters

- **name** (*str*) – Name of the function (defaults to function name).
- **num_params** (*int*) – Number of parameters the function accepts, or -1 for any number.

Decorator to register a user-defined scalar function.

Example:

```
@db.func('title_case')
def title_case(s):
    return s.title() if s else ''

# Usage:
title_case_books = Book.select(fn.title_case(Book.title))
```

register_window_function(*klass*[, *name=None*[, *num_params=-1*]])

Parameters

- **klass** – Class implementing window function API.
- **name** (*str*) – Window function name (defaults to name of class).
- **num_params** (*int*) – Number of parameters the function accepts, or -1 for any number.

Register a user-defined window function.

> **Attention:** This feature requires SQLite >= 3.25.0 **and** pysqlite3 >= 0.2.0.

The window function will be registered each time a new connection is opened. Additionally, if a connection is already open, the window function will be registered with the open connection.

window_function([*name=None*[, *num_params=-1*]])

Parameters

- **name** (*str*) – Name of the window function (defaults to class name).
- **num_params** (*int*) – Number of parameters the function accepts, or -1 for any number.

Class decorator to register a user-defined window function. Window functions must define the following methods:

- `step(<params>)` - receive values from a row and update state.
- `inverse(<params>)` - inverse of `step()` for the given values.
- `value()` - return the current value of the window function.
- `finalize()` - return the final value of the window function.

Example:

```
@db.window_function('my_sum')
class MySum(object):
    def __init__(self):
        self._value = 0

    def step(self, value):
        self._value += value

    def inverse(self, value):
        self._value -= value

    def value(self):
        return self._value

    def finalize(self):
        return self._value
```

table_function([*name=None*])

Class-decorator for registering a *TableFunction*. Table functions are user-defined functions that, rather than returning a single, scalar value, can return any number of rows of tabular data.

Example:

```
from playhouse.sqlite_ext import TableFunction

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

    def initialize(self, start=0, stop=None, step=1):
        """
        Table-functions declare an initialize() method, which is
        called with whatever arguments the user has called the
        function with.
        """
        self.start = self.current = start
        self.stop = stop or float('Inf')
        self.step = step

    def iterate(self, idx):
        """
        Iterate is called repeatedly by the SQLite database engine
        until the required number of rows has been read **or** the
        function raises a `StopIteration` signalling no more rows
        are available.
        """
        if self.current > self.stop:
            raise StopIteration

        ret, self.current = self.current, self.current + self.step
        return (ret,)

# Usage:
cursor = db.execute_sql('SELECT * FROM series(?, ?, ?)', (0, 5, 2))
for value, in cursor:
    print(value)
```

(continues on next page)

(continued from previous page)

```
# Prints:
# 0
# 2
# 4
```

unregister_aggregate(*name*)

Parameters **name** – Name of the user-defined aggregate function.

Unregister the user-defined aggregate function.

unregister_collation(*name*)

Parameters **name** – Name of the user-defined collation.

Unregister the user-defined collation.

unregister_function(*name*)

Parameters **name** – Name of the user-defined scalar function.

Unregister the user-defined scalar function.

unregister_table_function(*name*)

Parameters **name** – Name of the user-defined table function.

Returns True or False, depending on whether the function was removed.

Unregister the user-defined scalar function.

load_extension(*extension_module*)

Load the given C extension. If a connection is currently open in the calling thread, then the extension will be loaded for that connection as well as all subsequent connections.

For example, if you've compiled the closure table extension and wish to use it in your application, you might write:

```
db = SqliteExtDatabase('my_app.db')
db.load_extension('closure')
```

attach(*filename*, *name*)

Parameters

- **filename** (*str*) – Database to attach (or `:memory:` for in-memory)
- **name** (*str*) – Schema name for attached database.

Returns boolean indicating success

Register another database file that will be attached to every database connection. If the main database is currently connected, the new database will be attached on the open connection.

Note: Databases that are attached using this method will be attached every time a database connection is opened.

---

detach(*name*)

Parameters **name** (*str*) – Schema name for attached database.

Returns boolean indicating success

Unregister another database file that was attached previously with a call to `attach()`. If the main database is currently connected, the attached database will be detached from the open connection.

transaction(*[lock_type=None]*)

Parameters **lock_type** (*str*) – Locking strategy: DEFERRED, IMMEDIATE, EXCLUSIVE.

Create a transaction context-manager using the specified locking strategy (defaults to DEFERRED).

class PostgresqlDatabase(*database[, register_unicode=True[, encoding=None[, isolation_level=None]]]*)

Postgresql database implementation.

Additional optional keyword-parameters:

Parameters

- **register_unicode** (*bool*) – Register unicode types.
- **encoding** (*str*) – Database encoding.
- **isolation_level** (*int*) – Isolation level constant, defined in the `psycopg2.extensions` module.

set_time_zone(*timezone*)

Parameters **timezone** (*str*) – timezone name, e.g. "US/Central".

Returns no return value.

Set the timezone on the current connection. If no connection is open, then one will be opened.

class MySQLDatabase(*database[, **kwargs]*)

MySQL database implementation.

1.11.2 Query-builder

class Node

Base-class for all components which make up the AST for a SQL query.

static copy(*method*)

Decorator to use with Node methods that mutate the node's state. This allows method-chaining, e.g.:

```
query = MyModel.select()
new_query = query.where(MyModel.field == 'value')
```

unwrap()

API for recursively unwrapping "wrapped" nodes. Base case is to return self.

is_alias()

API for determining if a node, at any point, has been explicitly aliased by the user.

class Source(*[alias=None]*)

A source of row tuples, for example a table, join, or select query. By default provides a "magic" attribute named "c" that is a factory for column/attribute lookups, for example:

```
User = Table('users')
query = (User
         .select(User.c.username)
         .where(User.c.active == True)
         .order_by(User.c.username))
```

alias(*name*)
Returns a copy of the object with the given alias applied.

select(**columns*)

Parameters **columns** – *Column* instances, expressions, functions, sub-queries, or anything else that you would like to select.

Create a *Select* query on the table. If the table explicitly declares columns and no columns are provided, then by default all the table's defined columns will be selected.

join(*dest*[, *join_type='INNER'*[, *on=None*]])

Parameters

- **dest** (Source) – Join the table with the given destination.
- **join_type** (*str*) – Join type.
- **on** – Expression to use as join predicate.

Returns a *Join* instance.

Join type may be one of:

- JOIN.INNER
- JOIN.LEFT_OUTER
- JOIN.RIGHT_OUTER
- JOIN.FULL
- JOIN.FULL_OUTER
- JOIN.CROSS

left_outer_join(*dest*[, *on=None*])

Parameters

- **dest** (Source) – Join the table with the given destination.
- **on** – Expression to use as join predicate.

Returns a *Join* instance.

Convenience method for calling *join()* using a LEFT OUTER join.

class BaseTable
Base class for table-like objects, which support JOINs via operator overloading.

__and__(*dest*)
Perform an INNER join on dest.

__add__(*dest*)
Perform a LEFT OUTER join on dest.

__sub__(*dest*)
Perform a RIGHT OUTER join on dest.

__or__(*dest*)
Perform a FULL OUTER join on dest.

__mul__(*dest*)
Perform a CROSS join on dest.

class Table(*name*[, *columns=None*[, *primary_key=None*[, *schema=None*[, *alias=None*]]]])

Represents a table in the database (or a table-like object such as a view).

Parameters

- **name** (*str*) – Database table name
- **columns** (*tuple*) – List of column names (optional).
- **primary_key** (*str*) – Name of primary key column.
- **schema** (*str*) – Schema name used to access table (if necessary).
- **alias** (*str*) – Alias to use for table in SQL queries.

Note: If columns are specified, the magic "c" attribute will be disabled.

When columns are not explicitly defined, tables have a special attribute "c" which is a factory that provides access to table columns dynamically.

Example:

```
User = Table('users')
query = (User
         .select(User.c.id, User.c.username)
         .order_by(User.c.username))
```

Equivalent example when columns **are** specified:

```
User = Table('users', ('id', 'username'))
query = (User
         .select(User.id, User.username)
         .order_by(User.username))
```

bind([*database=None*])

Parameters **database** – *Database* object.

Bind this table to the given database (or unbind by leaving empty).

When a table is *bound* to a database, queries may be executed against it without the need to specify the database in the query's execute method.

bind_ctx([*database=None*])

Parameters **database** – *Database* object.

Return a context manager that will bind the table to the given database for the duration of the wrapped block.

select(**columns*)

Parameters **columns** – *Column* instances, expressions, functions, sub-queries, or anything else that you would like to select.

Create a *Select* query on the table. If the table explicitly declares columns and no columns are provided, then by default all the table's defined columns will be selected.

Example:

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

# Because columns were defined on the Table, we will default to
# selecting both of the User table's columns.
# Evaluates to SELECT id, username FROM users
query = User.select()

Note = Table('notes')
query = (Note
         .select(Note.c.content, Note.c.timestamp, User.username)
         .join(User, on=(Note.c.user_id == User.id))
         .where(Note.c.is_published == True)
         .order_by(Note.c.timestamp.desc()))

# Using a function to select users and the number of notes they
# have authored.
query = (User
         .select(
            User.username,
            fn.COUNT(Note.c.id).alias('n_notes'))
         .join(
            Note,
            JOIN.LEFT_OUTER,
            on=(User.id == Note.c.user_id))
         .order_by(fn.COUNT(Note.c.id).desc()))
```

insert(*[insert=None[, columns=None[, **kwargs]]]*)

Parameters

- **insert** – A dictionary mapping column to value, an iterable that yields dictionaries (i.e. list), or a *Select* query.
- **columns** (*list*) – The list of columns to insert into when the data being inserted is not a dictionary.
- **kwargs** – Mapping of column-name to value.

Create a *Insert* query into the table.

replace(*[insert=None[, columns=None[, **kwargs]]]*)

Parameters

- **insert** – A dictionary mapping column to value, an iterable that yields dictionaries (i.e. list), or a *Select* query.
- **columns** (*list*) – The list of columns to insert into when the data being inserted is not a dictionary.
- **kwargs** – Mapping of column-name to value.

Create a *Insert* query into the table whose conflict resolution method is to replace.

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

Parameters

- **update** – A dictionary mapping column to value.
- **kwargs** – Mapping of column-name to value.

Create a *Update* query for the table.

delete()
Create a *Delete* query for the table.

class Join(*lhs*, *rhs*[, *join_type=JOIN.INNER*[, *on=None*[, *alias=None*]]])
Represent a JOIN between to table-like objects.

Parameters

- **lhs** – Left-hand side of the join.
- **rhs** – Right-hand side of the join.
- **join_type** – Type of join. e.g. JOIN.INNER, JOIN.LEFT_OUTER, etc.
- **on** – Expression describing the join predicate.
- **alias** (*str*) – Alias to apply to joined data.

on(*predicate*)

Parameters **predicate** (*Expression*) – join predicate.

Specify the predicate expression used for this join.

class ValuesList(*values*[, *columns=None*[, *alias=None*]])
Represent a values list that can be used like a table.

Parameters

- **values** – a list-of-lists containing the row data to represent.
- **columns** (*list*) – the names to give to the columns in each row.
- **alias** (*str*) – alias to use for values-list.

Example:

```
data = [(1, 'first'), (2, 'second')]
vl = ValuesList(data, columns=('idx', 'name'))

query = (vl
         .select(vl.c.idx, vl.c.name)
         .order_by(vl.c.idx))
# Yields:
# SELECT t1.idx, t1.name
# FROM (VALUES (1, 'first'), (2, 'second')) AS t1(idx, name)
# ORDER BY t1.idx
```

columns(**names*)

Parameters **names** – names to apply to the columns of data.

Example:

```
vl = ValuesList([(1, 'first'), (2, 'second')])
vl = vl.columns('idx', 'name').alias('v')

query = vl.select(vl.c.idx, vl.c.name)
# Yields:
# SELECT v.idx, v.name
# FROM (VALUES (1, 'first'), (2, 'second')) AS v(idx, name)
```

class CTE(*name*, *query*[, *recursive=False*[, *columns=None*]])
Represent a common-table-expression. For example queries, see *Common Table Expressions*.

Parameters

- **name** – Name for the CTE.
- **query** – *Select* query describing CTE.
- **recursive** (*bool*) – Whether the CTE is recursive.
- **columns** (*list*) – Explicit list of columns produced by CTE (optional).

select_from(**columns*)

Create a SELECT query that utilizes the given common table expression as the source for a new query.

Parameters **columns** – One or more columns to select from the CTE.

Returns *Select* query utilizing the common table expression

union_all(*other*)

Used on the base-case CTE to construct the recursive term of the CTE.

Parameters **other** – recursive term, generally a *Select* query.

Returns a recursive *CTE* with the given recursive term.

class ColumnBase

Base-class for column-like objects, attributes or expressions.

Column-like objects can be composed using various operators and special methods.

- `&`: Logical AND
- `|`: Logical OR
- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `^`: Exclusive-OR
- `==`: Equality
- `!=`: Inequality
- `>`: Greater-than
- `<`: Less-than
- `>=`: Greater-than or equal
- `<=`: Less-than or equal
- `<<`: `IN`
- `>>`: `IS` (i.e. `IS NULL`)
- `%`: `LIKE`
- `**`: `ILIKE`
- `bin_and()`: Binary AND
- `bin_or()`: Binary OR
- `in_()`: `IN`
- `not_in()`: `NOT IN`

- `regexp()`: `REGEXP`
- `is_null(True/False)`: `IS NULL` or `IS NOT NULL`
- `contains(s)`: `LIKE %s%`
- `startswith(s)`: `LIKE s%`
- `endswith(s)`: `LIKE %s`
- `between(low, high)`: `BETWEEN low AND high`
- `concat()`: `||`

alias(*alias*)

Parameters **alias** (*str*) – Alias for the given column-like object.

Returns a *Alias* object.

Indicate the alias that should be given to the specified column-like object.

cast(*as_type*)

Parameters **as_type** (*str*) – Type name to cast to.

Returns a *Cast* object.

Create a `CAST` expression.

asc([*collation=None*[, *nulls=None*]])

Parameters

- **collation** (*str*) – Collation name to use for sorting.
- **nulls** (*str*) – Sort nulls (FIRST or LAST).

Returns an ascending *Ordering* object for the column.

desc([*collation=None*[, *nulls=None*]])

Parameters

- **collation** (*str*) – Collation name to use for sorting.
- **nulls** (*str*) – Sort nulls (FIRST or LAST).

Returns an descending *Ordering* object for the column.

__invert__()

Returns a *Negated* wrapper for the column.

class Column(*source*, *name*)

Parameters

- **source** (Source) – Source for column.
- **name** (*str*) – Column name.

Column on a table or a column returned by a sub-query.

class Alias(*node*, *alias*)

Parameters

- **node** (Node) – a column-like object.
- **alias** (*str*) – alias to assign to column.

Create a named alias for the given column-like object.

alias(*[alias=None]*)

Parameters **alias** (*str*) – new name (or None) for aliased column.

Create a new *Alias* for the aliased column-like object. If the new alias is None, then the original column-like object is returned.

class Negated(*node*)

Represents a negated column-like object.

class Value(*value*[, *converterNone*[, *unpack=True*]])

Parameters

- **value** – Python object or scalar value.
- **converter** – Function used to convert value into type the database understands.
- **unpack** (*bool*) – Whether lists or tuples should be unpacked into a list of values or treated as-is.

Value to be used in a parameterized query. It is the responsibility of the caller to ensure that the value passed in can be adapted to a type the database driver understands.

AsIs(*value*)

Represents a *Value* that is treated as-is, and passed directly back to the database driver. This may be useful if you are using database extensions that accept native Python data-types and you do not wish Peewee to impose any handling of the values.

class Cast(*node*, *cast*)

Parameters

- **node** – A column-like object.
- **cast** (*str*) – Type to cast to.

Represents a `CAST(<node> AS <cast>)` expression.

class Ordering(*node*, *direction*[, *collation=None*[, *nulls=None*]])

Parameters

- **node** – A column-like object.
- **direction** (*str*) – ASC or DESC
- **collation** (*str*) – Collation name to use for sorting.
- **nulls** (*str*) – Sort nulls (FIRST or LAST).

Represent ordering by a column-like object.

Postgresql supports a non-standard clause ("NULLS FIRST/LAST"). Peewee will automatically use an equivalent `CASE` statement for databases that do not support this (Sqlite / MySQL).

collate(*[collation=None]*)

Parameters **collation** (*str*) – Collation name to use for sorting.

Asc(*node*[, *collation=None*[, *nulls=None*]])

Short-hand for instantiating an ascending *Ordering* object.

Desc(*node*[, *collation=None*[, *nulls=None*]])

Short-hand for instantiating an descending *Ordering* object.

class Expression(*lhs*, *op*, *rhs*[, *flat=True*])

Parameters

- **lhs** – Left-hand side.
- **op** – Operation.
- **rhs** – Right-hand side.
- **flat** (*bool*) – Whether to wrap expression in parentheses.

Represent a binary expression of the form (lhs op rhs), e.g. (foo + 1).

class Entity(**path*)

Parameters **path** – Components that make up the dotted-path of the entity name.

Represent a quoted entity in a query, such as a table, column, alias. The name may consist of multiple components, e.g. "a_table"."column_name".

__getattr__(*self*, *attr*)
Factory method for creating sub-entities.

class SQL(*sql*[, *params=None*])

Parameters

- **sql** (*str*) – SQL query string.
- **params** (*tuple*) – Parameters for query (optional).

Represent a parameterized SQL query or query-fragment.

Check(*constraint*)

Parameters **constraint** (*str*) – Constraint SQL.

Represent a CHECK constraint.

class Function(*name*, *arguments*[, *coerce=True*[, *python_value=None*]])

Parameters

- **name** (*str*) – Function name.
- **arguments** (*tuple*) – Arguments to function.
- **coerce** (*bool*) – Whether to coerce the function result to a particular data-type when reading function return values from the cursor.
- **python_value** (*callable*) – Function to use for converting the return value from the cursor.

Represent an arbitrary SQL function call.

Note: Rather than instantiating this class directly, it is recommended to use the `fn` helper.

Example of using `fn` to call an arbitrary SQL function:

```
# Query users and count of tweets authored.
query = (User
         .select(User.username, fn.COUNT(Tweet.id).alias('ct'))
         .join(Tweet, JOIN.LEFT_OUTER, on=(User.id == Tweet.user_id))
         .group_by(User.username)
         .order_by(fn.COUNT(Tweet.id).desc()))
```

over([*partition_by=None*[, *order_by=None*[, *start=None*[, *end=None*[, *window=None*[, *exclude=None*]]]]])

Parameters

- **partition_by** (*list*) – List of columns to partition by.
- **order_by** (*list*) – List of columns / expressions to order window by.
- **start** – A *SQL* instance or a string expressing the start of the window range.
- **end** – A *SQL* instance or a string expressing the end of the window range.
- **frame_type** (*str*) – `Window.RANGE`, `Window.ROWS` or `Window.GROUPS`.
- **window** (`Window`) – A *Window* instance.
- **exclude** – Frame exclusion, one of `Window.CURRENT_ROW`, `Window.GROUP`, `Window.TIES` or `Window.NO_OTHERS`.

Note: For an in-depth guide to using window functions with Peewee, see the *Window functions* section.

Examples:

```
# Using a simple partition on a single column.
query = (Sample
         .select(
            Sample.counter,
            Sample.value,
            fn.AVG(Sample.value).over([Sample.counter]))
         .order_by(Sample.counter))

# Equivalent example Using a Window() instance instead.
window = Window(partition_by=[Sample.counter])
query = (Sample
         .select(
            Sample.counter,
            Sample.value,
            fn.AVG(Sample.value).over(window))
         .window(window)  # Note call to ".window()"
         .order_by(Sample.counter))

# Example using bounded window.
query = (Sample
         .select(Sample.value,
                 fn.SUM(Sample.value).over(
                    partition_by=[Sample.counter],
                    start=Window.CURRENT_ROW,  # current row
                    end=Window.following()))  # unbounded following
         .order_by(Sample.id))
```

filter(*where*)

Parameters **where** – Expression for filtering aggregate.

Add a `FILTER (WHERE...)` clause to an aggregate function. The where expression is evaluated to determine which rows are fed to the aggregate function. This SQL feature is supported for Postgres and SQLite.

coerce([*coerce=True*])

Parameters **coerce** (*bool*) – Whether to attempt to coerce function-call result to a Python data-type.

When coerce is `True`, the target data-type is inferred using several heuristics. Read the source for `BaseModelCursorWrapper._initialize_columns` method to see how this works.

python_value([*func=None*])

Parameters **python_value** (*callable*) – Function to use for converting the return value from the cursor.

Specify a particular function to use when converting values returned by the database cursor. For example:

```
# Get user and a list of their tweet IDs. The tweet IDs are
# returned as a comma-separated string by the db, so we'll split
# the result string and convert the values to python ints.
tweet_ids = (fn
             .GROUP_CONCAT(Tweet.id)
             .python_value(lambda idlist: [int(i) for i in idlist]))

query = (User
         .select(User.username, tweet_ids.alias('tweet_ids'))
         .group_by(User.username))

for user in query:
    print(user.username, user.tweet_ids)

# e.g.,
# huey [1, 4, 5, 7]
# mickey [2, 3, 6]
# zaizee []
```

fn()

The *fn()* helper is actually an instance of *Function* that implements a __getattr__ hook to provide a nice API for calling SQL functions.

To create a node representative of a SQL function call, use the function name as an attribute on `fn` and then provide the arguments as you would if calling a Python function:

```
# List users and the number of tweets they have authored,
# from highest-to-lowest:
sql_count = fn.COUNT(Tweet.id)
query = (User
         .select(User, sql_count.alias('count'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User)
         .order_by(sql_count.desc()))

# Get the timestamp of the most recent tweet:
query = Tweet.select(fn.MAX(Tweet.timestamp))
max_timestamp = query.scalar()  # Retrieve scalar result from query.
```

Function calls can, like anything else, be composed and nested:

```
# Get users whose username begins with "A" or "a":
a_users = User.select().where(fn.LOWER(fn.SUBSTR(User.username, 1, 1)) == 'a')
```

class Window([*partition_by=None*[, *order_by=None*[, *start=None*[, *end=None*[, *frame_type=None*[, *extends=None*[, *exclude=None*[, *alias=None*]]]]]]]])

Parameters

- **partition_by** (*list*) – List of columns to partition by.
- **order_by** (*list*) – List of columns to order by.
- **start** – A *SQL* instance or a string expressing the start of the window range.
- **end** – A *SQL* instance or a string expressing the end of the window range.
- **frame_type** (*str*) – `Window.RANGE`, `Window.ROWS` or `Window.GROUPS`.
- **extends** – A *Window* definition to extend. Alternately, you may specify the window's alias instead.
- **exclude** – Frame exclusion, one of `Window.CURRENT_ROW`, `Window.GROUP`, `Window.TIES` or `Window.NO_OTHERS`.
- **alias** (*str*) – Alias for the window.

Represent a WINDOW clause.

Note: For an in-depth guide to using window functions with Peewee, see the *Window functions* section.

RANGE

ROWS

GROUPS
Specify the window `frame_type`. See *Frame types: RANGE vs ROWS vs GROUPS*.

CURRENT_ROW
Reference to current row for use in start/end clause or the frame exclusion parameter.

NO_OTHERS

GROUP

TIES
Specify the window frame exclusion parameter.

static preceding([*value=None*])

Parameters **value** – Number of rows preceding. If `None` is UNBOUNDED.

Convenience method for generating SQL suitable for passing in as the `start` parameter for a window range.

static following([*value=None*])

Parameters **value** – Number of rows following. If `None` is UNBOUNDED.

Convenience method for generating SQL suitable for passing in as the `end` parameter for a window range.

as_rows()

as_range()

as_groups()
Specify the frame type.

extends([*window=None*])

Parameters **window** (`Window`) – A *Window* definition to extend. Alternately, you may specify the window's alias instead.

exclude([*frame_exclusion=None*])

Parameters **frame_exclusion** – Frame exclusion, one of `Window.CURRENT_ROW`, `Window.GROUP`, `Window.TIES` or `Window.NO_OTHERS`.

alias([*alias=None*])

Parameters **alias** (*str*) – Alias to use for window.

Case(*predicate, expression_tuples[, default=None]]*)

Parameters

- **predicate** – Predicate for CASE query (optional).
- **expression_tuples** – One or more cases to evaluate.
- **default** – Default value (optional).

Returns Representation of CASE statement.

Examples:

```
Number = Table('numbers', ('val',))

num_as_str = Case(Number.val, (
    (1, 'one'),
    (2, 'two'),
    (3, 'three')), 'a lot')

query = Number.select(Number.val, num_as_str.alias('num_str'))

# The above is equivalent to:
# SELECT "val",
#   CASE "val"
#       WHEN 1 THEN 'one'
#       WHEN 2 THEN 'two'
#       WHEN 3 THEN 'three'
#       ELSE 'a lot' END AS "num_str"
# FROM "numbers"

num_as_str = Case(None, (
    (Number.val == 1, 'one'),
    (Number.val == 2, 'two'),
    (Number.val == 3, 'three')), 'a lot')
query = Number.select(Number.val, num_as_str.alias('num_str'))

# The above is equivalent to:
# SELECT "val",
#   CASE
#       WHEN "val" = 1 THEN 'one'
#       WHEN "val" = 2 THEN 'two'
#       WHEN "val" = 3 THEN 'three'
#       ELSE 'a lot' END AS "num_str"
# FROM "numbers"
```

class NodeList(*nodes*[, *glue=' '*[, *parens=False*]])

Parameters

- **nodes** (*list*) – Zero or more nodes.
- **glue** (*str*) – How to join the nodes when converting to SQL.

- **parens** (*bool*) – Whether to wrap the resulting SQL in parentheses.

Represent a list of nodes, a multi-part clause, a list of parameters, etc.

CommaNodeList(*nodes*)

Parameters **nodes** (*list*) – Zero or more nodes.

Returns a *NodeList*

Represent a list of nodes joined by commas.

EnclosedNodeList(*nodes*)

Parameters **nodes** (*list*) – Zero or more nodes.

Returns a *NodeList*

Represent a list of nodes joined by commas and wrapped in parentheses.

class DQ(***query*)

Parameters **query** – Arbitrary filter expressions using Django-style lookups.

Represent a composable Django-style filter expression suitable for use with the *Model.filter()* or *ModelSelect.filter()* methods.

class Tuple(**args*)

Represent a SQL row value. Row-values are supported by most databases.

class OnConflict([*action=None*[, *update=None*[, *preserve=None*[, *where=None*[, *conflict_target=None*[, *conflict_where=None*[, *conflict_constraint=None*]]]]]]])

Parameters

- **action** (*str*) – Action to take when resolving conflict.
- **update** – A dictionary mapping column to new value.
- **preserve** – A list of columns whose values should be preserved from the original INSERT. See also *EXCLUDED*.
- **where** – Expression to restrict the conflict resolution.
- **conflict_target** – Column(s) that comprise the constraint.
- **conflict_where** – Expressions needed to match the constraint target if it is a partial index (index with a WHERE clause).
- **conflict_constraint** (*str*) – Name of constraint to use for conflict resolution. Currently only supported by Postgres.

Represent a conflict resolution clause for a data-modification query.

Depending on the database-driver being used, one or more of the above parameters may be required.

preserve(**columns*)

Parameters **columns** – Columns whose values should be preserved.

update([*_data=None*[, ***kwargs*]])

Parameters

- **_data** (*dict*) – Dictionary mapping column to new value.
- **kwargs** – Dictionary mapping column name to new value.

The `update()` method supports being called with either a dictionary of column-to-value, **or** keyword arguments representing the same.

where(**expressions*)

Parameters **expressions** – Expressions that restrict the action of the conflict resolution clause.

conflict_target(**constraints*)

Parameters **constraints** – Column(s) to use as target for conflict resolution.

conflict_where(**expressions*)

Parameters **expressions** – Expressions that match the conflict target index, in the case the conflict target is a partial index.

conflict_constraint(*constraint*)

Parameters **constraint** (*str*) – Name of constraints to use as target for conflict resolution. Currently only supported by Postgres.

class EXCLUDED

Helper object that exposes the `EXCLUDED` namespace that is used with `INSERT ... ON CONFLICT` to reference values in the conflicting data. This is a "magic" helper, such that one uses it by accessing attributes on it that correspond to a particular column.

Example:

```
class KV(Model):
    key = CharField(unique=True)
    value = IntegerField()

# Create one row.
KV.create(key='k1', value=1)

# Demonstrate usage of EXCLUDED.
# Here we will attempt to insert a new value for a given key. If that
# key already exists, then we will update its value with the *sum* of its
# original value and the value we attempted to insert -- provided that
# the new value is larger than the original value.
query = (KV.insert(key='k1', value=10)
         .on_conflict(conflict_target=[KV.key],
                      update={KV.value: KV.value + EXCLUDED.value},
                      where=(EXCLUDED.value > KV.value)))

# Executing the above query will result in the following data being
# present in the "kv" table:
# (key='k1', value=11)
query.execute()

# If we attempted to execute the query *again*, then nothing would be
# updated, as the new value (10) is now less than the value in the
# original row (11).
```

class BaseQuery

The parent class from which all other query classes are derived. While you will not deal with *BaseQuery* directly in your code, it implements some methods that are common across all query types.

default_row_type = ROW.DICT

bind([*database=None*])

Parameters **database** (Database) – Database to execute query against.

Bind the query to the given database for execution.

dicts([*as_dict=True*])

Parameters **as_dict** (*bool*) – Specify whether to return rows as dictionaries.

Return rows as dictionaries.

tuples([*as_tuples=True*])

Parameters **as_tuple** (*bool*) – Specify whether to return rows as tuples.

Return rows as tuples.

namedtuples([*as_namedtuple=True*])

Parameters **as_namedtuple** (*bool*) – Specify whether to return rows as named tuples.

Return rows as named tuples.

objects([*constructor=None*])

Parameters **constructor** – Function that accepts row dict and returns an arbitrary object.

Return rows as arbitrary objects using the given constructor.

sql()

Returns A 2-tuple consisting of the query's SQL and parameters.

execute(*database*)

Parameters **database** (Database) – Database to execute query against. Not required if query was previously bound to a database.

Execute the query and return result (depends on type of query being executed). For example, select queries the return result will be an iterator over the query results.

iterator([*database=None*])

Parameters **database** (Database) – Database to execute query against. Not required if query was previously bound to a database.

Execute the query and return an iterator over the result-set. For large result-sets this method is preferable as rows are not cached in-memory during iteration.

Note:

Because rows are not cached, the query may only be iterated over once. Subsequent iterations will return empty result-sets as the cursor will have been consumed.

Example:

```
query = StatTbl.select().order_by(StatTbl.timestamp).tuples()
for row in query.iterator(db):
    process_row(row)
```

__iter__()

Execute the query and return an iterator over the result-set.

Unlike *iterator()*, this method will cause rows to be cached in order to allow efficient iteration, indexing and slicing.

__getitem__(*value*)

Parameters **value** – Either an integer index or a slice.

Retrieve a row or range of rows from the result-set.

__len__()

Return the number of rows in the result-set.

> **Warning:** This does not issue a `COUNT()` query. Instead, the result-set is loaded as it would be during normal iteration, and the length is determined from the size of the result set.

class RawQuery([*sql=None*[, *params=None*[, ***kwargs*]]])

Parameters

- **sql** (*str*) – SQL query.
- **params** (*tuple*) – Parameters (optional).

Create a query by directly specifying the SQL to execute.

class Query([*where=None*[, *order_by=None*[, *limit=None*[, *offset=None*[, ***kwargs*]]]]])

Parameters

- **where** – Representation of WHERE clause.
- **order_by** (*tuple*) – Columns or values to order by.
- **limit** (*int*) – Value of LIMIT clause.
- **offset** (*int*) – Value of OFFSET clause.

Base-class for queries that support method-chaining APIs.

with_cte(**cte_list*)

Parameters **cte_list** – zero or more *CTE* objects.

Include the given common-table expressions in the query. Any previously specified CTEs will be overwritten. For examples of common-table expressions, see *Common Table Expressions*.

where(**expressions*)

Parameters **expressions** – zero or more expressions to include in the WHERE clause.

Include the given expressions in the WHERE clause of the query. The expressions will be AND-ed together with any previously-specified WHERE expressions.

Example selection users where the username is equal to 'somebody':

```
sq = User.select().where(User.username == 'somebody')
```

Example selecting tweets made by users who are either editors or administrators:

```
sq = Tweet.select().join(User).where(
    (User.is_editor == True) |
    (User.is_admin == True))
```

Example of deleting tweets by users who are no longer active:

```
inactive_users = User.select().where(User.active == False)
dq = (Tweet
      .delete()
      .where(Tweet.user.in_(inactive_users)))
dq.execute()  # Return number of tweets deleted.
```

Note: *where()* calls are chainable. Multiple calls will be "AND"-ed together.

orwhere(**expressions*)

Parameters **expressions** – zero or more expressions to include in the WHERE clause.

Include the given expressions in the WHERE clause of the query. This method is the same as the *Query.where()* method, except that the expressions will be OR-ed together with any previously-specified WHERE expressions.

order_by(**values*)

Parameters **values** – zero or more Column-like objects to order by.

Define the ORDER BY clause. Any previously-specified values will be overwritten.

order_by_extend(**values*)

Parameters **values** – zero or more Column-like objects to order by.

Extend any previously-specified ORDER BY clause with the given values.

limit(*value=None*)

Parameters **value** (*int*) – specify value for LIMIT clause.

offset(*value=None*)

Parameters **value** (*int*) – specify value for OFFSET clause.

paginate(*page*[, *paginate_by=20*])

Parameters

- **page** (*int*) – Page number of results (starting from 1).
- **paginate_by** (*int*) – Rows-per-page.

Convenience method for specifying the LIMIT and OFFSET in a more intuitive way.

This feature is designed with web-site pagination in mind, so the first page starts with `page=1`.

class SelectQuery

Select query helper-class that implements operator-overloads for creating compound queries.

cte(*name*[, *recursive=False*[, *columns=None*]])

Parameters

- **name** (*str*) – Alias for common table expression.
- **recursive** (*bool*) – Will this be a recursive CTE?
- **columns** (*list*) – List of column names (as strings).

Indicate that a query will be used as a common table expression. For example, if we are modelling a category tree and are using a parent-link foreign key, we can retrieve all categories and their absolute depths using a recursive CTE:

```
class Category(Model):
    name = TextField()
    parent = ForeignKeyField('self', backref='children', null=True)

# The base case of our recursive CTE will be categories that are at
# the root level -- in other words, categories without parents.
roots = (Category
         .select(Category.name, Value(0).alias('level'))
         .where(Category.parent.is_null())
         .cte(name='roots', recursive=True))

# The recursive term will select the category name and increment
# the depth, joining on the base term so that the recursive term
# consists of all children of the base category.
RTerm = Category.alias()
recursive = (RTerm
             .select(RTerm.name, (roots.c.level + 1).alias('level'))
             .join(roots, on=(RTerm.parent == roots.c.id)))

# Express <base term> UNION ALL <recursive term>.
cte = roots.union_all(recursive)

# Select name and level from the recursive CTE.
query = (cte
         .select_from(cte.c.name, cte.c.level)
         .order_by(cte.c.name))

for category in query:
    print(category.name, category.level)
```

For more examples of CTEs, see *Common Table Expressions*.

select_from(**columns*)

Parameters **columns** – one or more columns to select from the inner query.

Returns a new query that wraps the calling query.

Create a new query that wraps the current (calling) query. For example, suppose you have a simple `UNION` query, and need to apply an aggregation on the union result-set. To do this, you need to write something like:

```
SELECT "u"."owner", COUNT("u"."id") AS "ct"
FROM (
    SELECT "id", "owner", ... FROM "cars"
    UNION
    SELECT "id", "owner", ... FROM "motorcycles"
    UNION
    SELECT "id", "owner", ... FROM "boats") AS "u"
GROUP BY "u"."owner"
```

The *select_from()* method is designed to simplify constructing this type of query.

Example peewee code:

```
class Car(Model):
    owner = ForeignKeyField(Owner, backref='cars')
    # ... car-specific fields, etc ...
```

(continues on next page)

(continued from previous page)

```
class Motorcycle(Model):
    owner = ForeignKeyField(Owner, backref='motorcycles')
    # ... motorcycle-specific fields, etc ...

class Boat(Model):
    owner = ForeignKeyField(Owner, backref='boats')
    # ... boat-specific fields, etc ...

cars = Car.select(Car.owner)
motorcycles = Motorcycle.select(Motorcycle.owner)
boats = Boat.select(Boat.owner)

union = cars | motorcycles | boats

query = (union
         .select_from(union.c.owner, fn.COUNT(union.c.id))
         .group_by(union.c.owner))
```

union_all(*dest*)
Create a UNION ALL query with `dest`.

__add__(*dest*)
Create a UNION ALL query with `dest`.

union(*dest*)
Create a UNION query with `dest`.

__or__(*dest*)
Create a UNION query with `dest`.

intersect(*dest*)
Create an INTERSECT query with `dest`.

__and__(*dest*)
Create an INTERSECT query with `dest`.

except_(*dest*)
Create an EXCEPT query with `dest`. Note that the method name has a trailing "_" character since `except` is a Python reserved word.

__sub__(*dest*)
Create an EXCEPT query with `dest`.

class SelectBase
Base-class for *Select* and `CompoundSelect` queries.

peek(*database*[, *n=1*])

Parameters

- **database** (`Database`) – database to execute query against.
- **n** (*int*) – Number of rows to return.

Returns A single row if n = 1, else a list of rows.

Execute the query and return the given number of rows from the start of the cursor. This function may be called multiple times safely, and will always return the first N rows of results.

first(*database*[, *n=1*])

Parameters

- **database** (Database) – database to execute query against.
- **n** (*int*) – Number of rows to return.

Returns A single row if n = 1, else a list of rows.

Like the *peek()* method, except a LIMIT is applied to the query to ensure that only n rows are returned. Multiple calls for the same value of n will not result in multiple executions.

scalar(*database*[, *as_tuple=False*])

Parameters

- **database** (Database) – database to execute query against.
- **as_tuple** (*bool*) – Return the result as a tuple?

Returns Single scalar value if as_tuple = False, else row tuple.

Return a scalar value from the first row of results. If multiple scalar values are anticipated (e.g. multiple aggregations in a single query) then you may specify as_tuple=True to get the row tuple.

Example:

```
query = Note.select(fn.MAX(Note.timestamp))
max_ts = query.scalar(db)

query = Note.select(fn.MAX(Note.timestamp), fn.COUNT(Note.id))
max_ts, n_notes = query.scalar(db, as_tuple=True)
```

count(*database*[, *clear_limit=False*])

Parameters

- **database** (Database) – database to execute query against.
- **clear_limit** (*bool*) – Clear any LIMIT clause when counting.

Returns Number of rows in the query result-set.

Return number of rows in the query result-set.

Implemented by running SELECT COUNT(1) FROM (<current query>).

exists(*database*)

Parameters **database** (Database) – database to execute query against.

Returns Whether any results exist for the current query.

Return a boolean indicating whether the current query has any results.

get(*database*)

Parameters **database** (Database) – database to execute query against.

Returns A single row from the database or None.

Execute the query and return the first row, if it exists. Multiple calls will result in multiple queries being executed.

class CompoundSelectQuery(*lhs*, *op*, *rhs*)

Parameters

- **lhs** (SelectBase) – A Select or CompoundSelect query.
- **op** (*str*) – Operation (e.g. UNION, INTERSECT, EXCEPT).

- **rhs** (SelectBase) – A Select or CompoundSelect query.

Class representing a compound SELECT query.

class Select([*from_list=None*[, *columns=None*[, *group_by=None*[, *having=None*[, *distinct=None*[, *windows=None*[, *for_update=None*[, *for_update_of=None*[, *for_update_nowait=None*[, ***kwargs*]]]]]]]]]])

Parameters

- **from_list** (*list*) – List of sources for FROM clause.
- **columns** (*list*) – Columns or values to select.
- **group_by** (*list*) – List of columns or values to group by.
- **having** (Expression) – Expression for HAVING clause.
- **distinct** – Either a boolean or a list of column-like objects.
- **windows** (*list*) – List of *Window* clauses.
- **for_update** – Boolean or str indicating if SELECT...FOR UPDATE.
- **for_update_of** – One or more tables for FOR UPDATE OF clause.
- **for_update_nowait** (*bool*) – Specify NOWAIT locking.

Class representing a SELECT query.

Note: Rather than instantiating this directly, most-commonly you will use a factory method like *Table.select()* or *Model.select()*.

Methods on the select query can be chained together.

Example selecting some user instances from the database. Only the `id` and `username` columns are selected. When iterated, will return instances of the `User` model:

```
query = User.select(User.id, User.username)
for user in query:
    print(user.username)
```

Example selecting users and additionally the number of tweets made by the user. The `User` instances returned will have an additional attribute, 'count', that corresponds to the number of tweets made:

```
query = (User
         .select(User, fn.COUNT(Tweet.id).alias('count'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User))
for user in query:
    print(user.username, 'has tweeted', user.count, 'times')
```

Note: While it is possible to instantiate *Select* directly, more commonly you will build the query using the method-chaining APIs.

---

columns(**columns*)

Parameters **columns** – Zero or more column-like objects to SELECT.

Specify which columns or column-like values to SELECT.

select(**columns*)

Parameters **columns** – Zero or more column-like objects to SELECT.

Same as *Select.columns()*, provided for backwards-compatibility.

select_extend(**columns*)

Parameters **columns** – Zero or more column-like objects to SELECT.

Extend the current selection with the given columns.

Example:

```
def get_users(with_count=False):
    query = User.select()
    if with_count:
        query = (query
                 .select_extend(fn.COUNT(Tweet.id).alias('count'))
                 .join(Tweet, JOIN.LEFT_OUTER)
                 .group_by(User))
    return query
```

from_(**sources*)

Parameters **sources** – Zero or more sources for the FROM clause.

Specify which table-like objects should be used in the FROM clause.

```
User = Table('users')
Tweet = Table('tweets')
query = (User
         .select(User.c.username, Tweet.c.content)
         .from_(User, Tweet)
         .where(User.c.id == Tweet.c.user_id))
for row in query.execute(db):
    print(row['username'], '->', row['content'])
```

join(*dest*[, *join_type='INNER'*[, *on=None*]])

Parameters

- **dest** – A table or table-like object.
- **join_type** (*str*) – Type of JOIN, default is "INNER".
- **on** (Expression) – Join predicate.

Join type may be one of:

- JOIN.INNER
- JOIN.LEFT_OUTER
- JOIN.RIGHT_OUTER
- JOIN.FULL
- JOIN.FULL_OUTER
- JOIN.CROSS

Express a JOIN:

```
User = Table('users', ('id', 'username'))
Note = Table('notes', ('id', 'user_id', 'content'))

query = (Note
         .select(Note.content, User.username)
         .join(User, on=(Note.user_id == User.id)))
```

group_by(**columns*)

Parameters **values** – zero or more Column-like objects to group by.

Define the GROUP BY clause. Any previously-specified values will be overwritten.

Additionally, to specify all columns on a given table, you can pass the table/model object in place of the individual columns.

Example:

```
query = (User
         .select(User, fn.Count(Tweet.id).alias('count'))
         .join(Tweet)
         .group_by(User))
```

group_by_extend(**columns*)

Parameters **values** – zero or more Column-like objects to group by.

Extend the GROUP BY clause with the given columns.

having(**expressions*)

Parameters **expressions** – zero or more expressions to include in the HAVING clause.

Include the given expressions in the HAVING clause of the query. The expressions will be AND-ed together with any previously-specified HAVING expressions.

distinct(**columns*)

Parameters **columns** – Zero or more column-like objects.

Indicate whether this query should use a DISTINCT clause. By specifying a single value of `True` the query will use a simple SELECT DISTINCT. Specifying one or more columns will result in a SELECT DISTINCT ON.

window(**windows*)

Parameters **windows** – zero or more *Window* objects.

Define the WINDOW clause. Any previously-specified values will be overwritten.

Example:

```
# Equivalent example Using a Window() instance instead.
window = Window(partition_by=[Sample.counter])
query = (Sample
         .select(
            Sample.counter,
            Sample.value,
            fn.AVG(Sample.value).over(window))
         .window(window)  # Note call to ".window()"
         .order_by(Sample.counter))
```

for_update([*for_update=True*[, *of=None*[, *nowait=None*]]])

Parameters

- **for_update** – Either a boolean or a string indicating the desired expression, e.g. "FOR SHARE".
- **of** – One or more models to restrict locking to.
- **nowait** (*bool*) – Specify NOWAIT option when locking.

class _WriteQuery(*table*[, *returning=None*[, ***kwargs*]])

Parameters

- **table** (Table) – Table to write to.
- **returning** (*list*) – List of columns for RETURNING clause.

Base-class for write queries.

returning(**returning*)

Parameters **returning** – Zero or more column-like objects for RETURNING clause

Specify the RETURNING clause of query (if supported by your database).

```
query = (User
         .insert_many([{'username': 'foo'},
                       {'username': 'bar'},
                       {'username': 'baz'}])
         .returning(User.id, User.username)
         .namedtuples())
data = query.execute()
for row in data:
    print('added:', row.username, 'with id=', row.id)
```

class Update(*table*[, *update=None*[, ***kwargs*]])

Parameters

- **table** (Table) – Table to update.
- **update** (*dict*) – Data to update.

Class representing an UPDATE query.

Example:

```
PageView = Table('page_views')
query = (PageView
         .update({PageView.c.page_views: PageView.c.page_views + 1})
         .where(PageView.c.url == url))
query.execute(database)
```

from_(**sources*)

Parameters **sources** (Source) – one or more *Table*, *Model*, query, or *ValuesList* to join with.

Specify additional tables to join with using the UPDATE ... FROM syntax, which is supported by Postgres. The Postgres documentation provides additional detail, but to summarize:

> When a FROM clause is present, what essentially happens is that the target table is joined to the tables mentioned in the from_list, and each output row of the join represents an update operation for the target table. When using FROM you should ensure that the join produces at most one output row for each row to be modified.

Example:

```
# Update multiple users in a single query.
data = [('huey', True),
        ('mickey', False),
        ('zaizee', True)]
vl = ValuesList(data, columns=('username', 'is_admin'), alias='vl')

# Here we'll update the "is_admin" status of the above users,
# "joining" the VALUES() on the "username" column.
query = (User
         .update(is_admin=vl.c.is_admin)
         .from_(vl)
         .where(User.username == vl.c.username))
```

The above query produces the following SQL:

```
UPDATE "users" SET "is_admin" = "vl"."is_admin"
FROM (
    VALUES ('huey', t), ('mickey', f), ('zaizee', t))
    AS "vl"("username", "is_admin")
WHERE ("users"."username" = "vl"."username")
```

class Insert(*table*[, *insert=None*[, *columns=None*[, *on_conflict=None*[, ***kwargs*]]]])

Parameters

- **table** (Table) – Table to INSERT data into.
- **insert** – Either a dict, a list, or a query.
- **columns** (*list*) – List of columns when `insert` is a list or query.
- **on_conflict** – Conflict resolution strategy.

Class representing an INSERT query.

on_conflict_ignore([*ignore=True*])

Parameters **ignore** (*bool*) – Whether to add ON CONFLICT IGNORE clause.

Specify IGNORE conflict resolution strategy.

on_conflict_replace([*replace=True*])

Parameters **replace** (*bool*) – Whether to add ON CONFLICT REPLACE clause.

Specify REPLACE conflict resolution strategy.

on_conflict([*action=None*[, *update=None*[, *preserve=None*[, *where=None*[, *conflict_target=None*[, *conflict_where=None*[, *conflict_constraint=None*]]]]]]])

Parameters

- **action** (*str*) – Action to take when resolving conflict. If blank, action is assumed to be "update".
- **update** – A dictionary mapping column to new value.
- **preserve** – A list of columns whose values should be preserved from the original INSERT.
- **where** – Expression to restrict the conflict resolution.

- **conflict_target** – Column(s) that comprise the constraint.
- **conflict_where** – Expressions needed to match the constraint target if it is a partial index (index with a WHERE clause).
- **conflict_constraint** (*str*) – Name of constraint to use for conflict resolution. Currently only supported by Postgres.

Specify the parameters for an *OnConflict* clause to use for conflict resolution.

Examples:

```
class User(Model):
    username = TextField(unique=True)
    last_login = DateTimeField(null=True)
    login_count = IntegerField()

def log_user_in(username):
    now = datetime.datetime.now()

    # INSERT a new row for the user with the current timestamp and
    # login count set to 1. If the user already exists, then we
    # will preserve the last_login value from the "insert()" clause
    # and atomically increment the login-count.
    userid = (User
              .insert(username=username, last_login=now, login_count=1)
              .on_conflict(
                  conflict_target=[User.username],
                  preserve=[User.last_login],
                  update={User.login_count: User.login_count + 1})
              .execute())
    return userid
```

Example using the special *EXCLUDED* namespace:

```
class KV(Model):
    key = CharField(unique=True)
    value = IntegerField()

# Create one row.
KV.create(key='k1', value=1)

# Demonstrate usage of EXCLUDED.
# Here we will attempt to insert a new value for a given key. If that
# key already exists, then we will update its value with the *sum* of its
# original value and the value we attempted to insert -- provided that
# the new value is larger than the original value.
query = (KV.insert(key='k1', value=10)
         .on_conflict(conflict_target=[KV.key],
                      update={KV.value: KV.value + EXCLUDED.value},
                      where=(EXCLUDED.value > KV.value)))

# Executing the above query will result in the following data being
# present in the "kv" table:
# (key='k1', value=11)
query.execute()

# If we attempted to execute the query *again*, then nothing would be
# updated, as the new value (10) is now less than the value in the
# original row (11).
```

class Delete

Class representing a DELETE query.

class Index(*name*, *table*, *expressions*[, *unique=False*[, *safe=False*[, *where=None*[, *using=None*]]]])

Parameters

- **name** (*str*) – Index name.
- **table** (Table) – Table to create index on.
- **expressions** – List of columns to index on (or expressions).
- **unique** (*bool*) – Whether index is UNIQUE.
- **safe** (*bool*) – Whether to add IF NOT EXISTS clause.
- **where** (Expression) – Optional WHERE clause for index.
- **using** (*str*) – Index algorithm.

safe([*_safe=True*])

Parameters **_safe** (*bool*) – Whether to add IF NOT EXISTS clause.

where(**expressions*)

Parameters **expressions** – zero or more expressions to include in the WHERE clause.

Include the given expressions in the WHERE clause of the index. The expressions will be AND-ed together with any previously-specified WHERE expressions.

using([*_using=None*])

Parameters **_using** (*str*) – Specify index algorithm for USING clause.

class ModelIndex(*model*, *fields*[, *unique=False*[, *safe=True*[, *where=None*[, *using=None*[, *name=None*]]]]])

Parameters

- **model** (Model) – Model class to create index on.
- **fields** (*list*) – Fields to index.
- **unique** (*bool*) – Whether index is UNIQUE.
- **safe** (*bool*) – Whether to add IF NOT EXISTS clause.
- **where** (Expression) – Optional WHERE clause for index.
- **using** (*str*) – Index algorithm or type, e.g. 'BRIN', 'GiST' or 'GIN'.
- **name** (*str*) – Optional index name.

Expressive method for declaring an index on a model.

Examples:

```
class Article(Model):
    name = TextField()
    timestamp = TimestampField()
    status = IntegerField()
    flags = BitField()

    is_sticky = flags.flag(1)
    is_favorite = flags.flag(2)
```

(continues on next page)

(continued from previous page)

```
# CREATE INDEX ... ON "article" ("name", "timestamp")
idx = ModelIndex(Article, (Article.name, Article.timestamp))

# CREATE INDEX ... ON "article" ("name", "timestamp") WHERE "status" = 1
idx = idx.where(Article.status == 1)

# CREATE UNIQUE INDEX ... ON "article" ("timestamp" DESC, "flags" & 2) WHERE
↪"status" = 1
idx = ModelIndex(
    Article,
    (Article.timestamp.desc(), Article.flags.bin_and(2)),
    unique = True).where(Article.status == 1)
```

You can also use *Model.index()*:

```
idx = Article.index(Article.name, Article.timestamp).where(Article.status == 1)
```

To add an index to a model definition use *Model.add_index()*:

```
idx = Article.index(Article.name, Article.timestamp).where(Article.status == 1)

# Add above index definition to the model definition. When you call
# Article.create_table() (or database.create_tables([Article])), the
# index will be created.
Article.add_index(idx)
```

1.11.3 Fields

class Field([*null=False*[, *index=False*[, *unique=False*[, *column_name=None*[, *default=None*[, *primary_key=False*[, *constraints=None*[, *sequence=None*[, *collation=None*[, *unindexed=False*[, *choices=None*[, *help_text=None*[, *verbose_name=None*[, *index_type=None*]]]]]]]]]]]]]])

Parameters

- **null** (*bool*) – Field allows NULLs.
- **index** (*bool*) – Create an index on field.
- **unique** (*bool*) – Create a unique index on field.
- **column_name** (*str*) – Specify column name for field.
- **default** – Default value (enforced in Python, not on server).
- **primary_key** (*bool*) – Field is the primary key.
- **constraints** (*list*) – List of constraints to apply to column, for example: `[Check('price > 0')]`.
- **sequence** (*str*) – Sequence name for field.
- **collation** (*str*) – Collation name for field.
- **unindexed** (*bool*) – Declare field UNINDEXED (sqlite only).
- **choices** (*list*) – An iterable of 2-tuples mapping column values to display labels. Used for metadata purposes only, to help when displaying a dropdown of choices for field values, for example.

- **help_text** (*str*) – Help-text for field, metadata purposes only.
- **verbose_name** (*str*) – Verbose name for field, metadata purposes only.
- **index_type** (*str*) – Specify index type (postgres only), e.g. 'BRIN'.

Fields on a *Model* are analogous to columns on a table.

field_type = '<some field type>'
Attribute used to map this field to a column type, e.g. "INT". See the `FIELD` object in the source for more information.

column
Retrieve a reference to the underlying *Column* object.

model
The model the field is bound to.

name
The name of the field.

db_value(*value*)
Coerce a Python value into a value suitable for storage in the database. Sub-classes operating on special data-types will most likely want to override this method.

python_value(*value*)
Coerce a value from the database into a Python object. Sub-classes operating on special data-types will most likely want to override this method.

coerce(*value*)
This method is a shorthand that is used, by default, by both *db_value()* and *python_value()*.

Parameters **value** – arbitrary data from app or backend

Return type python data type

class IntegerField
Field class for storing integers.

class BigIntegerField
Field class for storing big integers (if supported by database).

class SmallIntegerField
Field class for storing small integers (if supported by database).

class AutoField
Field class for storing auto-incrementing primary keys.

Note: In SQLite, for performance reasons, the default primary key type simply uses the max existing value + 1 for new values, as opposed to the max ever value + 1. This means deleted records can have their primary keys reused. In conjunction with SQLite having foreign keys disabled by default (meaning ON DELETE is ignored, even if you specify it explicitly), this can lead to surprising and dangerous behaviour. To avoid this, you may want to use one or both of *AutoIncrementField* and `pragmas=[('foreign_keys', 'on')]` when you instantiate *SqliteDatabase*.

---

class BigAutoField
Field class for storing auto-incrementing primary keys using 64-bits.

class IdentityField([*generate_always=False*])

Parameters **generate_always** (*bool*) – if specified, then the identity will always be generated (and specifying the value explicitly during INSERT will raise a programming error). Otherwise, the identity value is only generated as-needed.

Field class for storing auto-incrementing primary keys using the new Postgres 10 *IDENTITY* column type. The column definition ends up looking like this:

```
id = IdentityField()
# "id" INT GENERATED BY DEFAULT AS IDENTITY NOT NULL PRIMARY KEY
```

Attention: Only supported by Postgres 10.0 and newer.

class FloatField
Field class for storing floating-point numbers.

class DoubleField
Field class for storing double-precision floating-point numbers.

class DecimalField([*max_digits=10*[, *decimal_places=5*[, *auto_round=False*[, *rounding=None*[, ***kwargs*]]]]])

Parameters

- **max_digits** (*int*) – Maximum digits to store.
- **decimal_places** (*int*) – Maximum precision.
- **auto_round** (*bool*) – Automatically round values.
- **rounding** – Defaults to `decimal.DefaultContext.rounding`.

Field class for storing decimal numbers. Values are represented as `decimal.Decimal` objects.

class CharField([*max_length=255*])
Field class for storing strings.

Note: Values that exceed length are not truncated automatically.

class FixedCharField
Field class for storing fixed-length strings.

Note: Values that exceed length are not truncated automatically.

class TextField
Field class for storing text.

class BlobField
Field class for storing binary data.

class BitField
Field class for storing options in a 64-bit integer column.

Usage:

```
class Post(Model):
    content = TextField()
    flags = BitField()

    is_favorite = flags.flag(1)
    is_sticky = flags.flag(2)
    is_minimized = flags.flag(4)
    is_deleted = flags.flag(8)

>>> p = Post()
>>> p.is_sticky = True
>>> p.is_minimized = True
>>> print(p.flags)  # Prints 4 | 2 --> "6"
6
>>> p.is_favorite
False
>>> p.is_sticky
True
```

We can use the flags on the Post class to build expressions in queries as well:

```
# Generates a WHERE clause that looks like:
# WHERE (post.flags & 1 != 0)
query = Post.select().where(Post.is_favorite)

# Query for sticky + favorite posts:
query = Post.select().where(Post.is_sticky & Post.is_favorite)
```

flag([*value=None*])

Parameters **value** (*int*) – Value associated with flag, typically a power of 2.

Returns a descriptor that can get or set specific bits in the overall value. When accessed on the class itself, it returns a *Expression* object suitable for use in a query.

If the value is not provided, it is assumed that each flag will be an increasing power of 2, so if you had four flags, they would have the values 1, 2, 4, 8.

class BigBitField

Field class for storing arbitrarily-large bitmaps in a BLOB. The field will grow the underlying buffer as necessary, ensuring there are enough bytes of data to support the number of bits of data being stored.

Example usage:

```
class Bitmap(Model):
    data = BigBitField()

bitmap = Bitmap()

# Sets the ith bit, e.g. the 1st bit, the 11th bit, the 63rd, etc.
bits_to_set = (1, 11, 63, 31, 55, 48, 100, 99)
for bit_idx in bits_to_set:
    bitmap.data.set_bit(bit_idx)

# We can test whether a bit is set using "is_set":
assert bitmap.data.is_set(11)
assert not bitmap.data.is_set(12)

# We can clear a bit:
```

(continues on next page)

(continued from previous page)

```
bitmap.data.clear_bit(11)
assert not bitmap.data.is_set(11)

# We can also "toggle" a bit. Recall that the 63rd bit was set earlier.
assert bitmap.data.toggle_bit(63) is False
assert bitmap.data.toggle_bit(63) is True
assert bitmap.data.is_set(63)
```

set_bit(*idx*)

Parameters **idx** (*int*) – Bit to set, indexed starting from zero.

Sets the *idx*-th bit in the bitmap.

clear_bit(*idx*)

Parameters **idx** (*int*) – Bit to clear, indexed starting from zero.

Clears the *idx*-th bit in the bitmap.

toggle_bit(*idx*)

Parameters **idx** (*int*) – Bit to toggle, indexed starting from zero.

Returns Whether the bit is set or not.

Toggles the *idx*-th bit in the bitmap and returns whether the bit is set or not.

Example:

```
>>> bitmap = Bitmap()
>>> bitmap.data.toggle_bit(10)  # Toggle the 10th bit.
True
>>> bitmap.data.toggle_bit(10)  # This will clear the 10th bit.
False
```

is_set(*idx*)

Parameters **idx** (*int*) – Bit index, indexed starting from zero.

Returns Whether the bit is set or not.

Returns boolean indicating whether the *idx*-th bit is set or not.

class UUIDField

Field class for storing `uuid.UUID` objects. With Postgres, the underlying column's data-type will be *UUID*. Since SQLite and MySQL do not have a native UUID type, the UUID is stored as a *VARCHAR* instead.

class BinaryUUIDField

Field class for storing `uuid.UUID` objects efficiently in 16-bytes. Uses the database's *BLOB* data-type (or *VARBINARY* in MySQL, or *BYTEA* in Postgres).

class DateTimeField([*formats=None*[, ***kwargs*]])

Parameters **formats** (*list*) – A list of format strings to use when coercing a string to a datetime.

Field class for storing `datetime.datetime` objects.

Accepts a special parameter `formats`, which contains a list of formats the datetime can be encoded with (for databases that do not have support for a native datetime data-type). The default supported formats are:

```
'%Y-%m-%d %H:%M:%S.%f' # year-month-day hour-minute-second.microsecond
'%Y-%m-%d %H:%M:%S' # year-month-day hour-minute-second
'%Y-%m-%d' # year-month-day
```

Note: SQLite does not have a native datetime data-type, so datetimes are stored as strings. This is handled transparently by Peewee, but if you have pre-existing data you should ensure it is stored as `YYYY-mm-dd HH:MM:SS` or one of the other supported formats.

year
Reference the year of the value stored in the column in a query.

```
Blog.select().where(Blog.pub_date.year == 2018)
```

month
Reference the month of the value stored in the column in a query.

day
Reference the day of the value stored in the column in a query.

hour
Reference the hour of the value stored in the column in a query.

minute
Reference the minute of the value stored in the column in a query.

second
Reference the second of the value stored in the column in a query.

to_timestamp()
Method that returns a database-specific function call that will allow you to work with the given date-time value as a numeric timestamp. This can sometimes simplify tasks like date math in a compatible way.

Example:

```
# Find all events that are exactly 1 hour long.
query = (Event
         .select()
         .where((Event.start.to_timestamp() + 3600) ==
                Event.stop.to_timestamp())
         .order_by(Event.start))
```

truncate(*date_part*)

Parameters **date_part** (*str*) – year, month, day, hour, minute or second.

Returns expression node to truncate date/time to given resolution.

Truncates the value in the column to the given part. This method is useful for finding all rows within a given month, for instance.

class DateField([*formats=None*[, ***kwargs*]])

Parameters **formats** (*list*) – A list of format strings to use when coercing a string to a date.

Field class for storing `datetime.date` objects.

Accepts a special parameter `formats`, which contains a list of formats the datetime can be encoded with (for databases that do not have support for a native date data-type). The default supported formats are:

```
'%Y-%m-%d' # year-month-day
'%Y-%m-%d %H:%M:%S' # year-month-day hour-minute-second
'%Y-%m-%d %H:%M:%S.%f' # year-month-day hour-minute-second.microsecond
```

Note: If the incoming value does not match a format, it is returned as-is.

year
Reference the year of the value stored in the column in a query.

```
Person.select().where(Person.dob.year == 1983)
```

month
Reference the month of the value stored in the column in a query.

day
Reference the day of the value stored in the column in a query.

to_timestamp()
See *DateTimeField.to_timestamp()*.

truncate(*date_part*)
See *DateTimeField.truncate()*. Note that only *year*, *month*, and *day* are meaningful for *DateField*.

class TimeField([*formats=None*[, ***kwargs*]])

Parameters **formats** (*list*) – A list of format strings to use when coercing a string to a time.

Field class for storing `datetime.time` objects (not `timedelta`).

Accepts a special parameter `formats`, which contains a list of formats the datetime can be encoded with (for databases that do not have support for a native time data-type). The default supported formats are:

```
'%H:%M:%S.%f' # hour:minute:second.microsecond
'%H:%M:%S' # hour:minute:second
'%H:%M' # hour:minute
'%Y-%m-%d %H:%M:%S.%f' # year-month-day hour-minute-second.microsecond
'%Y-%m-%d %H:%M:%S' # year-month-day hour-minute-second
```

Note: If the incoming value does not match a format, it is returned as-is.

hour
Reference the hour of the value stored in the column in a query.

```
evening_events = Event.select().where(Event.time.hour > 17)
```

minute
Reference the minute of the value stored in the column in a query.

second
Reference the second of the value stored in the column in a query.

class TimestampField([*resolution=1*[, *utc=False*[, ***kwargs*]]])

Parameters

- **resolution** – Can be provided as either a power of 10, or as an exponent indicating how many decimal places to store.
- **utc** (*bool*) – Treat timestamps as UTC.

Field class for storing date-times as integer timestamps. Sub-second resolution is supported by multiplying by a power of 10 to get an integer.

If the `resolution` parameter is `0` *or* `1`, then the timestamp is stored using second resolution. A resolution between `2` and `6` is treated as the number of decimal places, e.g. `resolution=3` corresponds to milliseconds. Alternatively, the decimal can be provided as a multiple of 10, such that `resolution=10` will store 1/10th of a second resolution.

The `resolution` parameter can be either 0-6 *or* 10, 100, etc up to 1000000 (for microsecond resolution). This allows sub-second precision while still using an *IntegerField* for storage. The default is second resolution.

Also accepts a boolean parameter `utc`, used to indicate whether the timestamps should be UTC. Default is `False`.

Finally, the field `default` is the current timestamp. If you do not want this behavior, then explicitly pass in `default=None`.

class IPField

Field class for storing IPv4 addresses efficiently (as integers).

class BooleanField

Field class for storing boolean values.

class BareField([*coerce=None*[, ***kwargs*]])

Parameters **coerce** – Optional function to use for converting raw values into a specific format.

Field class that does not specify a data-type (**SQLite-only**).

Since data-types are not enforced, you can declare fields without *any* data-type. It is also common for SQLite virtual tables to use meta-columns or untyped columns, so for those cases as well you may wish to use an untyped field.

Accepts a special `coerce` parameter, a function that takes a value coming from the database and converts it into the appropriate Python type.

class ForeignKeyField(*model*[, *field=None*[, *backref=None*[, *on_delete=None*[, *on_update=None*[, *deferrable=None*[, *object_id_name=None*[, *lazy_load=True*[, ***kwargs*]]]]]]]])

Parameters

- **model** (Model) – Model to reference or the string 'self' if declaring a self-referential foreign key.
- **field** (Field) – Field to reference on `model` (default is primary key).
- **backref** (*str*) – Accessor name for back-reference, or "+" to disable the back-reference accessor.
- **on_delete** (*str*) – ON DELETE action, e.g. `'CASCADE'`..
- **on_update** (*str*) – ON UPDATE action.
- **deferrable** (*str*) – Control when constraint is enforced, e.g. `'INITIALLY DEFERRED'`.
- **object_id_name** (*str*) – Name for object-id accessor.

- **lazy_load** (*bool*) – Fetch the related object when the foreign-key field attribute is accessed (if it was not already loaded). If this is disabled, accessing the foreign-key field will return the value stored in the foreign-key column.

Field class for storing a foreign key.

```
class User(Model):
    name = TextField()

class Tweet(Model):
    user = ForeignKeyField(User, backref='tweets')
    content = TextField()

# "user" attribute
>>> some_tweet.user
<User: charlie>

# "tweets" backref attribute
>>> for tweet in charlie.tweets:
...     print(tweet.content)
Some tweet
Another tweet
Yet another tweet
```

For an in-depth discussion of foreign-keys, joins and relationships between models, refer to *Relationships and Joins*.

Note: Foreign keys do not have a particular `field_type` as they will take their field type depending on the type of primary key on the model they are related to.

Note: If you manually specify a `field`, that field must be either a primary key or have a unique constraint.

Note: Take care with foreign keys in SQLite. By default, ON DELETE has no effect, which can have surprising (and usually unwanted) effects on your database integrity. This can affect you even if you don't specify `on_delete`, since the default ON DELETE behaviour (to fail without modifying your data) does not happen, and your data can be silently relinked. The safest thing to do is to specify `pragmas={'foreign_keys': 1}` when you instantiate *SqliteDatabase*.

---

class DeferredForeignKey(*rel_model_name*[, ***kwargs*])

Parameters **rel_model_name** (*str*) – Model name to reference.

Field class for representing a deferred foreign key. Useful for circular foreign-key references, for example:

```
class Husband(Model):
    name = TextField()
    wife = DeferredForeignKey('Wife', deferrable='INITIALLY DEFERRED')

class Wife(Model):
    name = TextField()
    husband = ForeignKeyField(Husband, deferrable='INITIALLY DEFERRED')
```

In the above example, when the `Wife` model is declared, the foreign-key `Husband.wife` is automatically resolved and turned into a regular *ForeignKeyField*.

Warning: *DeferredForeignKey* references are resolved when model classes are declared and created. This means that if you declare a *DeferredForeignKey* to a model class that has already been imported and created, the deferred foreign key instance will never be resolved. For example:

```
class User(Model):
    username = TextField()

class Tweet(Model):
    # This will never actually be resolved, because the User
    # model has already been declared.
    user = DeferredForeignKey('user', backref='tweets')
    content = TextField()
```

In cases like these you should use the regular *ForeignKeyField* *or* you can manually resolve deferred foreign keys like so:

```
# Tweet.user will be resolved into a ForeignKeyField:
DeferredForeignKey.resolve(User)
```

class ManyToManyField(*model*[, *backref=None*[, *through_model=None*[, *on_delete=None*[, *on_update=None*]]]])

Parameters

- **model** (Model) – Model to create relationship with.
- **backref** (*str*) – Accessor name for back-reference
- **through_model** (Model) – *Model* to use for the intermediary table. If not provided, a simple through table will be automatically created.
- **on_delete** (*str*) – ON DELETE action, e.g. `'CASCADE'`. Will be used for foreign-keys in through model.
- **on_update** (*str*) – ON UPDATE action. Will be used for foreign-keys in through model.

The *ManyToManyField* provides a simple interface for working with many-to-many relationships, inspired by Django. A many-to-many relationship is typically implemented by creating a junction table with foreign keys to the two models being related. For instance, if you were building a syllabus manager for college students, the relationship between students and courses would be many-to-many. Here is the schema using standard APIs:

Attention: This is not a field in the sense that there is no column associated with it. Rather, it provides a convenient interface for accessing rows of data related via a through model.

Standard way of declaring a many-to-many relationship (without the use of the *ManyToManyField*):

```
class Student(Model):
    name = CharField()

class Course(Model):
    name = CharField()

class StudentCourse(Model):
    student = ForeignKeyField(Student)
    course = ForeignKeyField(Course)
```

To query the courses for a particular student, you would join through the junction table:

```
# List the courses that "Huey" is enrolled in:
courses = (Course
           .select()
           .join(StudentCourse)
           .join(Student)
           .where(Student.name == 'Huey'))
for course in courses:
    print(course.name)
```

The *ManyToManyField* is designed to simplify this use-case by providing a *field-like* API for querying and modifying data in the junction table. Here is how our code looks using *ManyToManyField*:

```
class Student(Model):
    name = CharField()

class Course(Model):
    name = CharField()
    students = ManyToManyField(Student, backref='courses')
```

Note: It does not matter from Peewee's perspective which model the *ManyToManyField* goes on, since the back-reference is just the mirror image. In order to write valid Python, though, you will need to add the `ManyToManyField` on the second model so that the name of the first model is in the scope.

We still need a junction table to store the relationships between students and courses. This model can be accessed by calling the *get_through_model()* method. This is useful when creating tables.

```
# Create tables for the students, courses, and relationships between
# the two.
db.create_tables([
    Student,
    Course,
    Course.students.get_through_model()])
```

When accessed from a model instance, the *ManyToManyField* exposes a *ModelSelect* representing the set of related objects. Let's use the interactive shell to see how all this works:

```
>>> huey = Student.get(Student.name == 'huey')
>>> [course.name for course in huey.courses]
['English 101', 'CS 101']

>>> engl_101 = Course.get(Course.name == 'English 101')
>>> [student.name for student in engl_101.students]
['Huey', 'Mickey', 'Zaizee']
```

To add new relationships between objects, you can either assign the objects directly to the `ManyToManyField` attribute, or call the *add()* method. The difference between the two is that simply assigning will clear out any existing relationships, whereas `add()` can preserve existing relationships.

```
>>> huey.courses = Course.select().where(Course.name.contains('english'))
>>> for course in huey.courses.order_by(Course.name):
...     print course.name
English 101
English 151
English 201
English 221
```

(continues on next page)

(continued from previous page)

```
>>> cs_101 = Course.get(Course.name == 'CS 101')
>>> cs_151 = Course.get(Course.name == 'CS 151')
>>> huey.courses.add([cs_101, cs_151])
>>> [course.name for course in huey.courses.order_by(Course.name)]
['CS 101', 'CS151', 'English 101', 'English 151', 'English 201',
 'English 221']
```

This is quite a few courses, so let's remove the 200-level english courses. To remove objects, use the *remove()* method.

```
>>> huey.courses.remove(Course.select().where(Course.name.contains('2')))
2
>>> [course.name for course in huey.courses.order_by(Course.name)]
['CS 101', 'CS151', 'English 101', 'English 151']
```

To remove all relationships from a collection, you can use the `clear()` method. Let's say that English 101 is canceled, so we need to remove all the students from it:

```
>>> engl_101 = Course.get(Course.name == 'English 101')
>>> engl_101.students.clear()
```

Note: For an overview of implementing many-to-many relationships using standard Peewee APIs, check out the *Implementing Many to Many* section. For all but the most simple cases, you will be better off implementing many-to-many using the standard APIs.

---

through_model

The *Model* representing the many-to-many junction table. Will be auto-generated if not explicitly declared.

add(*value*[, *clear_existing=True*])

Parameters

- **value** – Either a *Model* instance, a list of model instances, or a *SelectQuery*.
- **clear_existing** (*bool*) – Whether to remove existing relationships.

Associate `value` with the current instance. You can pass in a single model instance, a list of model instances, or even a *ModelSelect*.

Example code:

```
# Huey needs to enroll in a bunch of courses, including all
# the English classes, and a couple Comp-Sci classes.
huey = Student.get(Student.name == 'Huey')

# We can add all the objects represented by a query.
english_courses = Course.select().where(
    Course.name.contains('english'))
huey.courses.add(english_courses)

# We can also add lists of individual objects.
cs101 = Course.get(Course.name == 'CS 101')
cs151 = Course.get(Course.name == 'CS 151')
huey.courses.add([cs101, cs151])
```

remove(*value*)

Parameters **value** – Either a *Model* instance, a list of model instances, or a *ModelSelect*.

Disassociate `value` from the current instance. Like *add()*, you can pass in a model instance, a list of model instances, or even a *ModelSelect*.

Example code:

```
# Huey is currently enrolled in a lot of english classes
# as well as some Comp-Sci. He is changing majors, so we
# will remove all his courses.
english_courses = Course.select().where(
    Course.name.contains('english'))
huey.courses.remove(english_courses)

# Remove the two Comp-Sci classes Huey is enrolled in.
cs101 = Course.get(Course.name == 'CS 101')
cs151 = Course.get(Course.name == 'CS 151')
huey.courses.remove([cs101, cs151])
```

clear()

Remove all associated objects.

Example code:

```
# English 101 is canceled this semester, so remove all
# the enrollments.
english_101 = Course.get(Course.name == 'English 101')
english_101.students.clear()
```

get_through_model()

Return the *Model* representing the many-to-many junction table. This can be specified manually when the field is being instantiated using the `through_model` parameter. If a `through_model` is not specified, one will automatically be created.

When creating tables for an application that uses *ManyToManyField*, **you must create the through table expicitly**.

```
# Get a reference to the automatically-created through table.
StudentCourseThrough = Course.students.get_through_model()

# Create tables for our two models as well as the through model.
db.create_tables([
    Student,
    Course,
    StudentCourseThrough])
```

class DeferredThroughModel

Place-holder for a through-model in cases where, due to a dependency, you cannot declare either a model or a many-to-many field without introducing NameErrors.

Example:

```
class Note(BaseModel):
    content = TextField()

NoteThroughDeferred = DeferredThroughModel()
```

(continues on next page)

(continued from previous page)

```
class User(BaseModel):
    username = TextField()
    notes = ManyToManyField(Note, through_model=NoteThroughDeferred)

# Cannot declare this before "User" since it has a foreign-key to
# the User model.
class NoteThrough(BaseModel):
    note = ForeignKeyField(Note)
    user = ForeignKeyField(User)

# Resolve dependencies.
NoteThroughDeferred.set_model(NoteThrough)
```

class CompositeKey(**field_names*)

Parameters **field_names** – Names of fields that comprise the primary key.

A primary key composed of multiple columns. Unlike the other fields, a composite key is defined in the model's `Meta` class after the fields have been defined. It takes as parameters the string names of the fields to use as the primary key:

```
class BlogTagThrough(Model):
    blog = ForeignKeyField(Blog, backref='tags')
    tag = ForeignKeyField(Tag, backref='blogs')

    class Meta:
        primary_key = CompositeKey('blog', 'tag')
```

1.11.4 Schema Manager

class SchemaManager(*model*[, *database=None*[, ***context_options*]])

Parameters

- **model** (`Model`) – Model class.
- **database** (`Database`) – If unspecified defaults to model._meta.database.

Provides methods for managing the creation and deletion of tables and indexes for the given model.

create_table([*safe=True*[, ***options*]])

Parameters

- **safe** (*bool*) – Specify IF NOT EXISTS clause.
- **options** – Arbitrary options.

Execute CREATE TABLE query for the given model.

drop_table([*safe=True*[, *drop_sequences=True*[, ***options*]]])

Parameters

- **safe** (*bool*) – Specify IF EXISTS clause.
- **drop_sequences** (*bool*) – Drop any sequences associated with the columns on the table (postgres only).
- **options** – Arbitrary options.

Execute DROP TABLE query for the given model.

truncate_table([*restart_identity=False*[, *cascade=False*]])

Parameters

- **restart_identity** (*bool*) – Restart the id sequence (postgres-only).
- **cascade** (*bool*) – Truncate related tables as well (postgres-only).

Execute TRUNCATE TABLE for the given model. If the database is Sqlite, which does not support TRUNCATE, then an equivalent DELETE query will be executed.

create_indexes([*safe=True*])

Parameters **safe** (*bool*) – Specify IF NOT EXISTS clause.

Execute CREATE INDEX queries for the indexes defined for the model.

drop_indexes([*safe=True*])

Parameters **safe** (*bool*) – Specify IF EXISTS clause.

Execute DROP INDEX queries for the indexes defined for the model.

create_sequence(*field*)

Parameters **field** (Field) – Field instance which specifies a sequence.

Create sequence for the given *Field*.

drop_sequence(*field*)

Parameters **field** (Field) – Field instance which specifies a sequence.

Drop sequence for the given *Field*.

create_foreign_key(*field*)

Parameters **field** (ForeignKeyField) – Foreign-key field constraint to add.

Add a foreign-key constraint for the given field. This method should not be necessary in most cases, as foreign-key constraints are created as part of table creation. The exception is when you are creating a circular foreign-key relationship using *DeferredForeignKey*. In those cases, it is necessary to first create the tables, then add the constraint for the deferred foreign-key:

```
class Language(Model):
    name = TextField()
    selected_snippet = DeferredForeignKey('Snippet')

class Snippet(Model):
    code = TextField()
    language = ForeignKeyField(Language, backref='snippets')

# Creates both tables but does not create the constraint for the
# Language.selected_snippet foreign key (because of the circular
# dependency).
db.create_tables([Language, Snippet])

# Explicitly create the constraint:
Language._schema.create_foreign_key(Language.selected_snippet)
```

For more information, see documentation on *Circular foreign key dependencies*.

> **Warning:** Because SQLite has limited support for altering existing tables, it is not possible to add a foreign-key constraint to an existing SQLite table.

create_all([*safe=True*[, ***table_options*]])

Parameters **safe** (*bool*) – Whether to specify IF NOT EXISTS.

Create sequence(s), index(es) and table for the model.

drop_all([*safe=True*[, *drop_sequences=True*[, ***options*]]])

Parameters

- **safe** (*bool*) – Whether to specify IF EXISTS.
- **drop_sequences** (*bool*) – Drop any sequences associated with the columns on the table (postgres only).
- **options** – Arbitrary options.

Drop table for the model and associated indexes.

1.11.5 Model

class Metadata(*model[, database=None[, table_name=None[, indexes=None[, primary_key=None[, constraints=None[, schema=None[, only_save_dirty=False[, depends_on=None[, options=None[, without_rowid=False[, **kwargs]]]]]]]]]]*)

Parameters

- **model** (Model) – Model class.
- **database** (Database) – database model is bound to.
- **table_name** (*str*) – Specify table name for model.
- **indexes** (*list*) – List of *ModelIndex* objects.
- **primary_key** – Primary key for model (only specified if this is a *CompositeKey* or False for no primary key.
- **constraints** (*list*) – List of table constraints.
- **schema** (*str*) – Schema table exists in.
- **only_save_dirty** (*bool*) – When *save()* is called, only save the fields which have been modified.
- **options** (*dict*) – Arbitrary options for the model.
- **without_rowid** (*bool*) – Specify WITHOUT ROWID (sqlite only).
- **kwargs** – Arbitrary setting attributes and values.

Store metadata for a *Model*.

This class should not be instantiated directly, but is instantiated using the attributes of a *Model* class' inner Meta class. Metadata attributes are then available on Model._meta.

table

Return a reference to the underlying *Table* object.

model_graph([*refs=True*[, *backrefs=True*[, *depth_first=True*]]])

Parameters

- **refs** (*bool*) – Follow foreign-key references.
- **backrefs** (*bool*) – Follow foreign-key back-references.
- **depth_first** (*bool*) – Do a depth-first search (`False` for breadth-first).

Traverse the model graph and return a list of 3-tuples, consisting of `(foreign key field, model class, is_backref)`.

set_database(*database*)

Parameters **database** (`Database`) – database object to bind Model to.

Bind the model class to the given *Database* instance.

> **Warning:** This API should not need to be used. Instead, to change a *Model* database at run-time, use one of the following:
>
> - *Model.bind()*
> - *Model.bind_ctx()* (bind for scope of a context manager).
> - *Database.bind()*
> - *Database.bind_ctx()*

set_table_name(*table_name*)

Parameters **table_name** (*str*) – table name to bind Model to.

Bind the model class to the given table name at run-time.

class SubclassAwareMetadata

Metadata subclass that tracks *Model* subclasses.

map_models(*fn*)

Apply a function to all subclasses.

class Model(***kwargs*)

Parameters **kwargs** – Mapping of field-name to value to initialize model with.

Model class provides a high-level abstraction for working with database tables. Models are a one-to-one mapping with a database table (or a table-like object, such as a view). Subclasses of `Model` declare any number of *Field* instances as class attributes. These fields correspond to columns on the table.

Table-level operations, such as *select()*, *update()*, *insert()* and *delete()* are implemented as classmethods. Row-level operations, such as *save()* and *delete_instance()* are implemented as instancemethods.

Example:

```
db = SqliteDatabase(':memory:')

class User(Model):
    username = TextField()
    join_date = DateTimeField(default=datetime.datetime.now)
    is_admin = BooleanField(default=False)

admin = User(username='admin', is_admin=True)
admin.save()
```

classmethod alias([*alias=None*])

Parameters **alias** (*str*) – Optional name for alias.

Returns *ModelAlias* instance.

Create an alias to the model-class. Model aliases allow you to reference the same *Model* multiple times in a query, for example when doing a self-join or sub-query.

Example:

```
Parent = Category.alias()
sq = (Category
      .select(Category, Parent)
      .join(Parent, on=(Category.parent == Parent.id))
      .where(Parent.name == 'parent category'))
```

classmethod select(**fields*)

Parameters **fields** – A list of model classes, field instances, functions or expressions. If no arguments are provided, all columns for the given model will be selected by default.

Returns *ModelSelect* query.

Create a SELECT query. If no fields are explicitly provided, the query will by default SELECT all the fields defined on the model, unless you are using the query as a sub-query, in which case only the primary key will be selected by default.

Example of selecting all columns:

```
query = User.select().where(User.active == True).order_by(User.username)
```

Example of selecting all columns on *Tweet* and the parent model, *User*. When the `user` foreign key is accessed on a *Tweet* instance no additional query will be needed (see *N+1* for more details):

```
query = (Tweet
         .select(Tweet, User)
         .join(User)
         .order_by(Tweet.created_date.desc()))

for tweet in query:
    print(tweet.user.username, '->', tweet.content)
```

Example of subquery only selecting the primary key:

```
inactive_users = User.select().where(User.active == False)

# Here, instead of defaulting to all columns, Peewee will default
# to only selecting the primary key.
Tweet.delete().where(Tweet.user.in_(inactive_users)).execute()
```

classmethod update([*__data=None*[, ***update*]])

Parameters

- **__data** (*dict*) – `dict` of fields to values.
- **update** – Field-name to value mapping.

Create an UPDATE query.

Example showing users being marked inactive if their registration has expired:

```
q = (User
     .update({User.active: False})
     .where(User.registration_expired == True))
q.execute()  # Execute the query, returning number of rows updated.
```

Example showing an atomic update:

```
q = (PageView
     .update({PageView.count: PageView.count + 1})
     .where(PageView.url == url))
q.execute()  # Execute the query.
```

Note: When an update query is executed, the number of rows modified will be returned.

classmethod insert([*__data=None*[, ***insert*]])

Parameters

- **__data** (*dict*) – `dict` of fields to values to insert.
- **insert** – Field-name to value mapping.

Create an INSERT query.

Insert a new row into the database. If any fields on the model have default values, these values will be used if the fields are not explicitly set in the `insert` dictionary.

Example showing creation of a new user:

```
q = User.insert(username='admin', active=True, registration_expired=False)
q.execute()  # perform the insert.
```

You can also use *Field* objects as the keys:

```
new_id = User.insert({User.username: 'admin'}).execute()
```

If you have a model with a default value on one of the fields, and that field is not specified in the `insert` parameter, the default will be used:

```
class User(Model):
    username = CharField()
    active = BooleanField(default=True)

# This INSERT query will automatically specify `active=True`:
User.insert(username='charlie')
```

Note: When an insert query is executed on a table with an auto-incrementing primary key, the primary key of the new row will be returned.

classmethod insert_many(*rows*[, *fields=None*])

Parameters

- **rows** – An iterable that yields rows to insert.
- **fields** (*list*) – List of fields being inserted.

Returns number of rows modified (see note).

INSERT multiple rows of data.

The `rows` parameter must be an iterable that yields dictionaries or tuples, where the ordering of the tuple values corresponds to the fields specified in the `fields` argument. As with *insert()*, fields that are not specified in the dictionary will use their default value, if one exists.

Note: Due to the nature of bulk inserts, each row must contain the same fields. The following will not work:

```
Person.insert_many([
    {'first_name': 'Peewee', 'last_name': 'Herman'},
    {'first_name': 'Huey'},  # Missing "last_name"!
]).execute()
```

Example of inserting multiple Users:

```
data = [
    ('charlie', True),
    ('huey', False),
    ('zaizee', False)]
query = User.insert_many(data, fields=[User.username, User.is_admin])
query.execute()
```

Equivalent example using dictionaries:

```
data = [
    {'username': 'charlie', 'is_admin': True},
    {'username': 'huey', 'is_admin': False},
    {'username': 'zaizee', 'is_admin': False}]

# Insert new rows.
User.insert_many(data).execute()
```

Because the `rows` parameter can be an arbitrary iterable, you can also use a generator:

```
def get_usernames():
    for username in ['charlie', 'huey', 'peewee']:
        yield {'username': username}
User.insert_many(get_usernames()).execute()
```

Warning: If you are using SQLite, your SQLite library must be version 3.7.11 or newer to take advantage of bulk inserts.

Note: SQLite has a default limit of 999 bound variables per statement. This limit can be modified at compile-time or at run-time, **but** if modifying at run-time, you can only specify a *lower* value than the default limit.

For more information, check out the following SQLite documents:

- Max variable number limit
- Changing run-time limits
- SQLite compile-time flags

Note: The default return value is the number of rows modified. However, when using Postgres, Peewee will return a cursor by default that yields the primary-keys of the inserted rows. To disable this functionality with Postgres, use an empty call to `returning()`.

classmethod insert_from(*query*, *fields*)

Parameters

- **query** (`Select`) – SELECT query to use as source of data.
- **fields** – Fields to insert data into.

Returns number of rows modified (see note).

INSERT data using a SELECT query as the source. This API should be used for queries of the form *INSERT INTO ... SELECT FROM ...*.

Example of inserting data across tables for denormalization purposes:

```
source = (User
          .select(User.username, fn.COUNT(Tweet.id))
          .join(Tweet, JOIN.LEFT_OUTER)
          .group_by(User.username))

UserTweetDenorm.insert_from(
    source,
    [UserTweetDenorm.username, UserTweetDenorm.num_tweets]).execute()
```

Note: The default return value is the number of rows modified. However, when using Postgres, Peewee will return a cursor by default that yields the primary-keys of the inserted rows. To disable this functionality with Postgres, use an empty call to `returning()`.

classmethod replace([*__data=None*[, ***insert*]])

Parameters

- **__data** (*dict*) – `dict` of fields to values to insert.
- **insert** – Field-name to value mapping.

Create an INSERT query that uses REPLACE for conflict-resolution.

See *Model.insert()* for examples.

classmethod replace_many(*rows*[, *fields=None*])

Parameters

- **rows** – An iterable that yields rows to insert.
- **fields** (*list*) – List of fields being inserted.

INSERT multiple rows of data using REPLACE for conflict-resolution.

See *Model.insert_many()* for examples.

classmethod raw(*sql*, **params*)

Parameters

- **sql** (*str*) – SQL query to execute.
- **params** – Parameters for query.

Execute a SQL query directly.

Example selecting rows from the User table:

```
q = User.raw('select id, username from users')
for user in q:
    print(user.id, user.username)
```

Note: Generally the use of `raw` is reserved for those cases where you can significantly optimize a select query. It is useful for select queries since it will return instances of the model.

classmethod delete()

Create a DELETE query.

Example showing the deletion of all inactive users:

```
q = User.delete().where(User.active == False)
q.execute()  # Remove the rows, return number of rows removed.
```

Warning: This method performs a delete on the *entire table*. To delete a single instance, see *Model.delete_instance()*.

classmethod create(***query*)

Parameters **query** – Mapping of field-name to value.

INSERT new row into table and return corresponding model instance.

Example showing the creation of a user (a row will be added to the database):

```
user = User.create(username='admin', password='test')
```

Note: The create() method is a shorthand for instantiate-then-save.

classmethod bulk_create(*model_list*[, *batch_size=None*])

Parameters

- **model_list** (*iterable*) – a list or other iterable of unsaved *Model* instances.
- **batch_size** (*int*) – number of rows to batch per insert. If unspecified, all models will be inserted in a single query.

Returns no return value.

Efficiently INSERT multiple unsaved model instances into the database. Unlike *insert_many()*, which accepts row data as a list of either dictionaries or lists, this method accepts a list of unsaved model instances.

Example:

```
# List of 10 unsaved users.
user_list = [User(username='u%s' % i) for i in range(10)]

# All 10 users are inserted in a single query.
User.bulk_create(user_list)
```

Batches:

```
user_list = [User(username='u%s' % i) for i in range(10)]

with database.atomic():
    # Will execute 4 INSERT queries (3 batches of 3, 1 batch of 1).
    User.bulk_create(user_list, batch_size=3)
```

Warning:

- The primary-key value for the newly-created models will only be set if you are using Postgresql (which supports the `RETURNING` clause).
- SQLite generally has a limit of 999 bound parameters for a query, so the batch size should be roughly 1000 / number-of-fields.
- When a batch-size is provided it is **strongly recommended** that you wrap the call in a transaction or savepoint using *Database.atomic()*. Otherwise an error in a batch mid-way through could leave the database in an inconsistent state.

classmethod bulk_update(*model_list*, *fields*[, *batch_size=None*])

Parameters

- **model_list** (*iterable*) – a list or other iterable of *Model* instances.
- **fields** (*list*) – list of fields to update.
- **batch_size** (*int*) – number of rows to batch per insert. If unspecified, all models will be inserted in a single query.

Returns total number of rows updated.

Efficiently UPDATE multiple model instances.

Example:

```
# First, create 3 users.
u1, u2, u3 = [User.create(username='u%s' % i) for i in (1, 2, 3)]

# Now let's modify their usernames.
u1.username = 'u1-x'
u2.username = 'u2-y'
u3.username = 'u3-z'

# Update all three rows using a single UPDATE query.
User.bulk_update([u1, u2, u3], fields=[User.username])
```

If you have a large number of objects to update, it is strongly recommended that you specify a `batch_size` and wrap the operation in a transaction:

```
with database.atomic():
    User.bulk_update(user_list, fields=['username'], batch_size=50)
```

Warning:

- SQLite generally has a limit of 999 bound parameters for a query.
- When a batch-size is provided it is **strongly recommended** that you wrap the call in a transaction or savepoint using *Database.atomic()*. Otherwise an error in a batch mid-way through could leave the database in an inconsistent state.

classmethod get(**query*, ***filters*)

Parameters

- **query** – Zero or more *Expression* objects.
- **filters** – Mapping of field-name to value for Django-style filter.

Raises DoesNotExist

Returns Model instance matching the specified filters.

Retrieve a single model instance matching the given filters. If no model is returned, a DoesNotExist is raised.

```
user = User.get(User.username == username, User.active == True)
```

This method is also exposed via the *SelectQuery*, though it takes no parameters:

```
active = User.select().where(User.active == True)
try:
    user = active.where(
        (User.username == username) &
        (User.active == True)
    ).get()
except User.DoesNotExist:
    user = None
```

Note: The *get()* method is shorthand for selecting with a limit of 1. It has the added behavior of raising an exception when no matching row is found. If more than one row is found, the first row returned by the database cursor will be used.

classmethod get_or_none(**query*, ***filters*)

Identical to *Model.get()* but returns None if no model matches the given filters.

classmethod get_by_id(*pk*)

Parameters **pk** – Primary-key value.

Short-hand for calling *Model.get()* specifying a lookup by primary key. Raises a DoesNotExist if instance with the given primary key value does not exist.

Example:

```
user = User.get_by_id(1)  # Returns user with id = 1.
```

classmethod set_by_id(*key*, *value*)

Parameters

- **key** – Primary-key value.

- **value** (*dict*) – Mapping of field to value to update.

Short-hand for updating the data with the given primary-key. If no row exists with the given primary key, no exception will be raised.

Example:

```
# Set "is_admin" to True on user with id=3.
User.set_by_id(3, {'is_admin': True})
```

classmethod delete_by_id(*pk*)

Parameters **pk** – Primary-key value.

Short-hand for deleting the row with the given primary-key. If no row exists with the given primary key, no exception will be raised.

classmethod get_or_create(***kwargs*)

Parameters

- **kwargs** – Mapping of field-name to value.
- **defaults** – Default values to use if creating a new row.

Returns Tuple of *Model* instance and boolean indicating if a new object was created.

Attempt to get the row matching the given filters. If no matching row is found, create a new row.

Warning: Race-conditions are possible when using this method.

Example **without** `get_or_create`:

```
# Without `get_or_create`, we might write:
try:
    person = Person.get(
        (Person.first_name == 'John') &
        (Person.last_name == 'Lennon'))
except Person.DoesNotExist:
    person = Person.create(
        first_name='John',
        last_name='Lennon',
        birthday=datetime.date(1940, 10, 9))
```

Equivalent code using `get_or_create`:

```
person, created = Person.get_or_create(
    first_name='John',
    last_name='Lennon',
    defaults={'birthday': datetime.date(1940, 10, 9)})
```

classmethod filter(**dq_nodes*, ***filters*)

Parameters

- **dq_nodes** – Zero or more *DQ* objects.
- **filters** – Django-style filters.

Returns *ModelSelect* query.

get_id()

Returns The primary-key of the model instance.

save([*force_insert=False*[, *only=None*]])

Parameters

- **force_insert** (*bool*) – Force INSERT query.
- **only** (*list*) – Only save the given *Field* instances.

Returns Number of rows modified.

Save the data in the model instance. By default, the presence of a primary-key value will cause an UPDATE query to be executed.

Example showing saving a model instance:

```
user = User()
user.username = 'some-user'  # does not touch the database
user.save()  # change is persisted to the db
```

dirty_fields

Return list of fields that have been modified.

Return type list

Note: If you just want to persist modified fields, you can call `model.save(only=model.dirty_fields)`.

If you **always** want to only save a model's dirty fields, you can use the Meta option `only_save_dirty = True`. Then, any time you call *Model.save()*, by default only the dirty fields will be saved, e.g.

```
class Person(Model):
    first_name = CharField()
    last_name = CharField()
    dob = DateField()

    class Meta:
        database = db
        only_save_dirty = True
```

Warning: Peewee determines whether a field is "dirty" by observing when the field attribute is set on a model instance. If the field contains a value that is mutable, such as a dictionary instance, and that dictionary is then modified, Peewee will not notice the change.

is_dirty()

Return boolean indicating whether any fields were manually set.

delete_instance([*recursive=False*[, *delete_nullable=False*]])

Parameters

- **recursive** (*bool*) – Delete related models.
- **delete_nullable** (*bool*) – Delete related models that have a null foreign key. If `False` nullable relations will be set to NULL.

Delete the given instance. Any foreign keys set to cascade on delete will be deleted automatically. For more programmatic control, you can specify `recursive=True`, which will delete any non-nullable related

models (those that *are* nullable will be set to NULL). If you wish to delete all dependencies regardless of whether they are nullable, set `delete_nullable=True`.

example:

```
some_obj.delete_instance()  # it is gone forever
```

classmethod bind(*database*[, *bind_refs=True*[, *bind_backrefs=True*]])

Parameters

- **database** (*Database*) – database to bind to.
- **bind_refs** (*bool*) – Bind related models.
- **bind_backrefs** (*bool*) – Bind back-reference related models.

Bind the model (and specified relations) to the given database.

See also: *Database.bind()*.

classmethod bind_ctx(*database*[, *bind_refs=True*[, *bind_backrefs=True*]])

Like *bind()*, but returns a context manager that only binds the models for the duration of the wrapped block.

See also: *Database.bind_ctx()*.

classmethod table_exists()

Returns boolean indicating whether the table exists.

classmethod create_table([*safe=True*[, ***options*]])

Parameters **safe** (*bool*) – If set to `True`, the create table query will include an `IF NOT EXISTS` clause.

Create the model table, indexes, constraints and sequences.

Example:

```
with database:
    SomeModel.create_table()  # Execute the create table query.
```

classmethod drop_table([*safe=True*[, ***options*]])

Parameters **safe** (*bool*) – If set to `True`, the create table query will include an `IF EXISTS` clause.

Drop the model table.

truncate_table([*restart_identity=False*[, *cascade=False*]])

Parameters

- **restart_identity** (*bool*) – Restart the id sequence (postgres-only).
- **cascade** (*bool*) – Truncate related tables as well (postgres-only).

Truncate (delete all rows) for the model.

classmethod index(**fields*[, *unique=False*[, *safe=True*[, *where=None*[, *using=None*[, *name=None*]]]]])

Parameters

- **fields** – Fields to index.
- **unique** (*bool*) – Whether index is UNIQUE.

- **safe** (*bool*) – Whether to add IF NOT EXISTS clause.
- **where** (Expression) – Optional WHERE clause for index.
- **using** (*str*) – Index algorithm.
- **name** (*str*) – Optional index name.

Expressive method for declaring an index on a model. Wraps the declaration of a *ModelIndex* instance.

Examples:

```
class Article(Model):
    name = TextField()
    timestamp = TimestampField()
    status = IntegerField()
    flags = BitField()

    is_sticky = flags.flag(1)
    is_favorite = flags.flag(2)

# CREATE INDEX ... ON "article" ("name", "timestamp" DESC)
idx = Article.index(Article.name, Article.timestamp.desc())

# Be sure to add the index to the model:
Article.add_index(idx)

# CREATE UNIQUE INDEX ... ON "article" ("timestamp" DESC, "flags" & 2)
# WHERE ("status" = 1)
idx = (Article
       .index(Article.timestamp.desc(),
              Article.flags.bin_and(2),
              unique=True)
       .where(Article.status == 1))

# Add index to model:
Article.add_index(idx)
```

classmethod add_index(**args*, ***kwargs*)

Parameters

- **args** – a *ModelIndex* instance, Field(s) to index, or a *SQL* instance that contains the SQL for creating the index.
- **kwargs** – Keyword arguments passed to *ModelIndex* constructor.

Add an index to the model's definition.

Note: This method does not actually create the index in the database. Rather, it adds the index definition to the model's metadata, so that a subsequent call to *create_table()* will create the new index (along with the table).

Examples:

```
class Article(Model):
    name = TextField()
    timestamp = TimestampField()
    status = IntegerField()
```

(continues on next page)

(continued from previous page)

```
    flags = BitField()

    is_sticky = flags.flag(1)
    is_favorite = flags.flag(2)

# CREATE INDEX ... ON "article" ("name", "timestamp") WHERE "status" = 1
idx = Article.index(Article.name, Article.timestamp).where(Article.status ==␣
↪1)
Article.add_index(idx)

# CREATE UNIQUE INDEX ... ON "article" ("timestamp" DESC, "flags" & 2)
ts_flags_idx = Article.index(
    Article.timestamp.desc(),
    Article.flags.bin_and(2),
    unique=True)
Article.add_index(ts_flags_idx)

# You can also specify a list of fields and use the same keyword
# arguments that the ModelIndex constructor accepts:
Article.add_index(
    Article.name,
    Article.timestamp.desc(),
    where=(Article.status == 1))

# Or even specify a SQL query directly:
Article.add_index(SQL('CREATE INDEX ...'))
```

dependencies(*[search_nullable=False]*)

Parameters **search_nullable** (*bool*) – Search models related via a nullable foreign key

Return type Generator expression yielding queries and foreign key fields.

Generate a list of queries of dependent models. Yields a 2-tuple containing the query and corresponding foreign key field. Useful for searching dependencies of a model, i.e. things that would be orphaned in the event of a delete.

__iter__()

Returns a *ModelSelect* for the given class.

Convenience function for iterating over all instances of a model.

Example:

```
Setting.insert_many([
    {'key': 'host', 'value': '192.168.1.2'},
    {'key': 'port': 'value': '1337'},
    {'key': 'user': 'value': 'nuggie'}]).execute()

# Load settings from db into dict.
settings = {setting.key: setting.value for setting in Setting}
```

__len__()

Returns Count of rows in table.

Example:

```
n_accounts = len(Account)

# Is equivalent to:
n_accounts = Account.select().count()
```

class ModelAlias(*model*[, *alias=None*])

Parameters

- **model** (Model) – Model class to reference.
- **alias** (*str*) – (optional) name for alias.

Provide a separate reference to a model in a query.

class ModelSelect(*model*, *fields_or_models*)

Parameters

- **model** (Model) – Model class to select.
- **fields_or_models** – List of fields or model classes to select.

Model-specific implementation of SELECT query.

switch([*ctx=None*])

Parameters **ctx** – A *Model*, *ModelAlias*, subquery, or other object that was joined-on.

Switch the *join context* - the source which subsequent calls to *join()* will be joined against. Used for specifying multiple joins against a single table.

If the `ctx` is not given, then the query's model will be used.

The following example selects from tweet and joins on both user and tweet-flag:

```
sq = Tweet.select().join(User).switch(Tweet).join(TweetFlag)

# Equivalent (since Tweet is the query's model)
sq = Tweet.select().join(User).switch().join(TweetFlag)
```

objects([*constructor=None*])

Parameters **constructor** – Constructor (defaults to returning model instances)

Return result rows as objects created using the given constructor. The default behavior is to create model instances.

Note: This method can be used, when selecting field data from multiple sources/models, to make all data available as attributes on the model being queried (as opposed to constructing the graph of joined model instances). For very complex queries this can have a positive performance impact, especially iterating large result sets.

Similarly, you can use *dicts()*, *tuples()* or *namedtuples()* to achieve even more performance.

---

join(*dest*[, *join_type='INNER'*[, *on=None*[, *src=None*[, *attr=None*]]]])

Parameters

- **dest** – A *Model*, *ModelAlias*, *Select* query, or other object to join to.
- **join_type** (*str*) – Join type, defaults to INNER.
- **on** – Join predicate or a *ForeignKeyField* to join on.

- **src** – Explicitly specify the source of the join. If not specified then the current *join context* will be used.
- **attr** (*str*) – Attribute to use when projecting columns from the joined model.

Join with another table-like object.

Join type may be one of:

- `JOIN.INNER`
- `JOIN.LEFT_OUTER`
- `JOIN.RIGHT_OUTER`
- `JOIN.FULL`
- `JOIN.FULL_OUTER`
- `JOIN.CROSS`

Example selecting tweets and joining on user in order to restrict to only those tweets made by "admin" users:

```
sq = Tweet.select().join(User).where(User.is_admin == True)
```

Example selecting users and joining on a particular foreign key field. See the *example app* for a real-life usage:

```
sq = User.select().join(Relationship, on=Relationship.to_user)
```

For an in-depth discussion of foreign-keys, joins and relationships between models, refer to *Relationships and Joins*.

join_from(*src*, *dest*[, *join_type='INNER'*[, *on=None*[, *attr=None*]]])

Parameters

- **src** – Source for join.
- **dest** – Table to join to.

Use same parameter order as the non-model-specific *join()*. Bypasses the *join context* by requiring the join source to be specified.

filter(**args*, ***kwargs*)

Parameters

- **args** – Zero or more *DQ* objects.
- **kwargs** – Django-style keyword-argument filters.

Use Django-style filters to express a WHERE clause.

prefetch(**subqueries*)

Parameters **subqueries** – A list of *Model* classes or select queries to prefetch.

Returns a list of models with selected relations prefetched.

Execute the query, prefetching the given additional resources.

See also *prefetch()* standalone function.

Example:

```
# Fetch all Users and prefetch their associated tweets.
query = User.select().prefetch(Tweet)
for user in query:
    print(user.username)
    for tweet in user.tweets:
        print('  *', tweet.content)
```

Note: Because `prefetch` must reconstruct a graph of models, it is necessary to be sure that the foreign-key/primary-key of any related models are selected, so that the related objects can be mapped correctly.

prefetch(*sq*, **subqueries*)

Parameters

- **sq** – Query to use as starting-point.
- **subqueries** – One or more models or *ModelSelect* queries to eagerly fetch.

Returns a list of models with selected relations prefetched.

Eagerly fetch related objects, allowing efficient querying of multiple tables when a 1-to-many relationship exists.

For example, it is simple to query a many-to-1 relationship efficiently:

```
query = (Tweet
         .select(Tweet, User)
         .join(User))
for tweet in query:
    # Looking up tweet.user.username does not require a query since
    # the related user's columns were selected.
    print(tweet.user.username, '->', tweet.content)
```

To efficiently do the inverse, query users and their tweets, you can use prefetch:

```
query = User.select()
for user in prefetch(query, Tweet):
    print(user.username)
    for tweet in user.tweets:  # Does not require additional query.
        print('    ', tweet.content)
```

Note: Because `prefetch` must reconstruct a graph of models, it is necessary to be sure that the foreign-key/primary-key of any related models are selected, so that the related objects can be mapped correctly.

1.11.6 Query-builder Internals

class AliasManager

Manages the aliases assigned to *Source* objects in SELECT queries, so as to avoid ambiguous references when multiple sources are used in a single query.

add(*source*)

Add a source to the AliasManager's internal registry at the current scope. The alias will be automatically generated using the following scheme (where each level of indentation refers to a new scope):

Parameters **source** (*Source*) – Make the manager aware of a new source. If the source has already been added, the call is a no-op.

get(*source*[, *any_depth=False*])
Return the alias for the source in the current scope. If the source does not have an alias, it will be given the next available alias.

Parameters **source** (Source) – The source whose alias should be retrieved.

Returns The alias already assigned to the source, or the next available alias.

Return type str

__setitem__(*source*, *alias*)
Manually set the alias for the source at the current scope.

Parameters **source** (Source) – The source for which we set the alias.

push()
Push a new scope onto the stack.

pop()
Pop scope from the stack.

class State(*scope*[, *parentheses=False*[, *subquery=False*[, ***kwargs*]]])
Lightweight object for representing the state at a given scope. During SQL generation, each object visited by the *Context* can inspect the state. The *State* class allows Peewee to do things like:

- Use a common interface for field types or SQL expressions, but use vendor-specific data-types or operators.
- Compile a *Column* instance into a fully-qualified attribute, as a named alias, etc, depending on the value of the `scope`.
- Ensure parentheses are used appropriately.

Parameters

- **scope** (*int*) – The scope rules to be applied while the state is active.
- **parentheses** (*bool*) – Wrap the contained SQL in parentheses.
- **subquery** (*bool*) – Whether the current state is a child of an outer query.
- **kwargs** (*dict*) – Arbitrary settings which should be applied in the current state.

class Context(***settings*)
Converts Peewee structures into parameterized SQL queries.

Peewee structures should all implement a *__sql__* method, which will be called by the *Context* class during SQL generation. The *__sql__* method accepts a single parameter, the *Context* instance, which allows for recursive descent and introspection of scope and state.

scope
Return the currently-active scope rules.

parentheses
Return whether the current state is wrapped in parentheses.

subquery
Return whether the current state is the child of another query.

scope_normal([***kwargs*])
The default scope. Sources are referred to by alias, columns by dotted-path from the source.

scope_source([***kwargs*])
Scope used when defining sources, e.g. in the column list and FROM clause of a SELECT query. This scope is used for defining the fully-qualified name of the source and assigning an alias.

scope_values([**kwargs])
Scope used for UPDATE, INSERT or DELETE queries, where instead of referencing a source by an alias, we refer to it directly. Similarly, since there is a single table, columns do not need to be referenced by dotted-path.

scope_cte([**kwargs])
Scope used when generating the contents of a common-table-expression. Used after a WITH statement, when generating the definition for a CTE (as opposed to merely a reference to one).

scope_column([**kwargs])
Scope used when generating SQL for a column. Ensures that the column is rendered with it's correct alias. Was needed because when referencing the inner projection of a sub-select, Peewee would render the full SELECT query as the "source" of the column (instead of the query's alias + . + column). This scope allows us to avoid rendering the full query when we only need the alias.

sql(*obj*)
Append a composable Node object, sub-context, or other object to the query AST. Python values, such as integers, strings, floats, etc. are treated as parameterized values.

Returns The updated Context object.

literal(*keyword*)
Append a string-literal to the current query AST.

Returns The updated Context object.

parse(*node*)

Parameters **node** (Node) – Instance of a Node subclass.

Returns a 2-tuple consisting of (sql, parameters).

Convert the given node to a SQL AST and return a 2-tuple consisting of the SQL query and the parameters.

query()

Returns a 2-tuple consisting of (sql, parameters) for the context.

1.11.7 Constants and Helpers

class Proxy
Create a proxy or placeholder for another object.

initialize(*obj*)

Parameters **obj** – Object to proxy to.

Bind the proxy to the given object. Afterwards all attribute lookups and method calls on the proxy will be sent to the given object.

Any callbacks that have been registered will be called.

attach_callback(*callback*)

Parameters **callback** – A function that accepts a single parameter, the bound object.

Returns self

Add a callback to be executed when the proxy is initialized.

class DatabaseProxy
Proxy subclass that is suitable to use as a placeholder for a Database instance.

See *Dynamically defining a database* for details on usage.

chunked(*iterable*, *n*)

Parameters

- **iterable** – an iterable that is the source of the data to be chunked.
- **n** (*int*) – chunk size

Returns a new iterable that yields *n*-length chunks of the source data.

Efficient implementation for breaking up large lists of data into smaller-sized chunks.

Usage:

```
it = range(10)  # An iterable that yields 0...9.

# Break the iterable into chunks of length 4.
for chunk in chunked(it, 4):
    print(', '.join(str(num) for num in chunk))

# PRINTS:
# 0, 1, 2, 3
# 4, 5, 6, 7
# 8, 9
```

1.12 SQLite Extensions

The default *SqliteDatabase* already includes many SQLite-specific features:

- *General notes on using SQLite*.
- *Configuring SQLite using PRAGMA statements*.
- *User-defined functions, aggregate and collations*.
- *Locking modes for transactions*.

The `playhouse.sqlite_ext` includes even more SQLite features, including:

- *Full-text search*
- *JSON extension integration*
- *Closure table extension support*
- *LSM1 extension support*
- *User-defined table functions*
- Support for online backups using backup API: *backup_to_file()*
- *BLOB API support, for efficient binary data storage*.
- *Additional helpers*, including bloom filter, more.

1.12.1 Getting started

To get started with the features described in this document, you will want to use the *SqliteExtDatabase* class from the `playhouse.sqlite_ext` module. Furthermore, some features require the `playhouse._sqlite_ext` C extension – these features will be noted in the documentation.

Instantiating a *SqliteExtDatabase*:

```
from playhouse.sqlite_ext import SqliteExtDatabase

db = SqliteExtDatabase('my_app.db', pragmas=(
    ('cache_size', -1024 * 64),  # 64MB page-cache.
    ('journal_mode', 'wal'),  # Use WAL-mode (you should always use this!).
    ('foreign_keys', 1))  # Enforce foreign-key constraints.
```

1.12.2 APIs

class SqliteExtDatabase(*database*[, *pragmas=None*[, *timeout=5*[, *c_extensions=None*[, *rank_functions=True*[, *hash_functions=False*[, *regexp_function=False*[, *bloomfilter=False*]]]]]]])

Parameters

- **pragmas** (*list*) – A list of 2-tuples containing pragma key and value to set every time a connection is opened.
- **timeout** – Set the busy-timeout on the SQLite driver (in seconds).
- **c_extensions** (*bool*) – Declare that C extension speedups must/must-not be used. If set to `True` and the extension module is not available, will raise an `ImproperlyConfigured` exception.
- **rank_functions** (*bool*) – Make search result ranking functions available.
- **hash_functions** (*bool*) – Make hashing functions available (md5, sha1, etc).
- **regexp_function** (*bool*) – Make the REGEXP function available.
- **bloomfilter** (*bool*) – Make the *bloom filter* available.

Extends *SqliteDatabase* and inherits methods for declaring user-defined functions, pragmas, etc.

class CSqliteExtDatabase(*database*[, *pragmas=None*[, *timeout=5*[, *c_extensions=None*[, *rank_functions=True*[, *hash_functions=False*[, *regexp_function=False*[, *bloomfilter=False*[, *replace_busy_handler=False*]]]]]]]])

Parameters

- **pragmas** (*list*) – A list of 2-tuples containing pragma key and value to set every time a connection is opened.
- **timeout** – Set the busy-timeout on the SQLite driver (in seconds).
- **c_extensions** (*bool*) – Declare that C extension speedups must/must-not be used. If set to `True` and the extension module is not available, will raise an `ImproperlyConfigured` exception.
- **rank_functions** (*bool*) – Make search result ranking functions available.
- **hash_functions** (*bool*) – Make hashing functions available (md5, sha1, etc).
- **regexp_function** (*bool*) – Make the REGEXP function available.
- **bloomfilter** (*bool*) – Make the *bloom filter* available.
- **replace_busy_handler** (*bool*) – Use a smarter busy-handler implementation.

Extends *SqliteExtDatabase* and requires that the `playhouse._sqlite_ext` extension module be available.

on_commit(*fn*)

Register a callback to be executed whenever a transaction is committed on the current connection. The callback accepts no parameters and the return value is ignored.

However, if the callback raises a `ValueError`, the transaction will be aborted and rolled-back.

Example:

```
db = CSqliteExtDatabase(':memory:')

@db.on_commit
def on_commit():
    logger.info('COMMITing changes')
```

on_rollback(*fn*)

Register a callback to be executed whenever a transaction is rolled back on the current connection. The callback accepts no parameters and the return value is ignored.

Example:

```
@db.on_rollback
def on_rollback():
    logger.info('Rolling back changes')
```

on_update(*fn*)

Register a callback to be executed whenever the database is written to (via an *UPDATE*, *INSERT* or *DELETE* query). The callback should accept the following parameters:

- `query` - the type of query, either *INSERT*, *UPDATE* or *DELETE*.
- database name - the default database is named *main*.
- table name - name of table being modified.
- rowid - the rowid of the row being modified.

The callback's return value is ignored.

Example:

```
db = CSqliteExtDatabase(':memory:')

@db.on_update
def on_update(query_type, db, table, rowid):
    # e.g. INSERT row 3 into table users.
    logger.info('%s row %s into table %s', query_type, rowid, table)
```

changes()

Return the number of rows modified in the currently-open transaction.

autocommit

Property which returns a boolean indicating if autocommit is enabled. By default, this value will be `True` except when inside a transaction (or *`atomic()`* block).

Example:

```
>>> db = CSqliteExtDatabase(':memory:')
>>> db.autocommit
True
>>> with db.atomic():
...     print(db.autocommit)
```

(continues on next page)

(continued from previous page)

```
...
False
>>> db.autocommit
True
```

backup(*destination*[, *pages=None*, *name=None*, *progress=None*])

Parameters

- **destination** (SqliteDatabase) – Database object to serve as destination for the backup.
- **pages** (*int*) – Number of pages per iteration. Default value of -1 indicates all pages should be backed-up in a single step.
- **name** (*str*) – Name of source database (may differ if you used ATTACH DATABASE to load multiple databases). Defaults to "main".
- **progress** – Progress callback, called with three parameters: the number of pages remaining, the total page count, and whether the backup is complete.

Example:

```
master = CSqliteExtDatabase('master.db')
replica = CSqliteExtDatabase('replica.db')

# Backup the contents of master to replica.
master.backup(replica)
```

backup_to_file(*filename*[, *pages*, *name*, *progress*])

Parameters

- **filename** – Filename to store the database backup.
- **pages** (*int*) – Number of pages per iteration. Default value of -1 indicates all pages should be backed-up in a single step.
- **name** (*str*) – Name of source database (may differ if you used ATTACH DATABASE to load multiple databases). Defaults to "main".
- **progress** – Progress callback, called with three parameters: the number of pages remaining, the total page count, and whether the backup is complete.

Backup the current database to a file. The backed-up data is not a database dump, but an actual SQLite database file.

Example:

```
db = CSqliteExtDatabase('app.db')

def nightly_backup():
    filename = 'backup-%s.db' % (datetime.date.today())
    db.backup_to_file(filename)
```

blob_open(*table*, *column*, *rowid*[, *read_only=False*])

Parameters

- **table** (*str*) – Name of table containing data.
- **column** (*str*) – Name of column containing data.

- **rowid** (*int*) – ID of row to retrieve.
- **read_only** (*bool*) – Open the blob for reading only.

Returns *Blob* instance which provides efficient access to the underlying binary data.

Return type *Blob*

See *Blob* and *ZeroBlob* for more information.

Example:

```
class Image(Model):
    filename = TextField()
    data = BlobField()

buf_size = 1024 * 1024 * 8  # Allocate 8MB for storing file.
rowid = Image.insert({Image.filename: 'thefile.jpg',
                      Image.data: ZeroBlob(buf_size)}).execute()

# Open the blob, returning a file-like object.
blob = db.blob_open('image', 'data', rowid)

# Write some data to the blob.
blob.write(image_data)
img_size = blob.tell()

# Read the data back out of the blob.
blob.seek(0)
image_data = blob.read(img_size)
```

class RowIDField

Primary-key field that corresponds to the SQLite `rowid` field. For more information, see the SQLite documentation on rowid tables..

Example:

```
class Note(Model):
    rowid = RowIDField()  # Will be primary key.
    content = TextField()
    timestamp = TimestampField()
```

class DocIDField

Subclass of *RowIDField* for use on virtual tables that specifically use the convention of `docid` for the primary key. As far as I know this only pertains to tables using the FTS3 and FTS4 full-text search extensions.

> **Attention:** In FTS3 and FTS4, "docid" is simply an alias for "rowid". To reduce confusion, it's probably best to just always use *RowIDField* and never use *DocIDField*.

```
class NoteIndex(FTSModel):
    docid = DocIDField()  # "docid" is used as an alias for "rowid".
    content = SearchField()

    class Meta:
        database = db
```

class AutoIncrementField

SQLite, by default, may reuse primary key values after rows are deleted. To ensure that the primary key is

always monotonically increasing, regardless of deletions, you should use *AutoIncrementField*. There is a small performance cost for this feature. For more information, see the SQLite docs on autoincrement.

class JSONField(*json_dumps=None*, *json_loads=None*, ...)

Field class suitable for storing JSON data, with special methods designed to work with the json1 extension.

SQLite 3.9.0 added JSON support in the form of an extension library. The SQLite json1 extension provides a number of helper functions for working with JSON data. These APIs are exposed as methods of a special field-type, *JSONField*.

To access or modify specific object keys or array indexes in a JSON structure, you can treat the *JSONField* as if it were a dictionary/list.

Parameters

- **json_dumps** – (optional) function for serializing data to JSON strings. If not provided, will use the stdlib `json.dumps`.
- **json_loads** – (optional) function for de-serializing JSON to Python objects. If not provided, will use the stdlib `json.loads`.

Note: To customize the JSON serialization or de-serialization, you can specify a custom `json_dumps` and `json_loads` callables. These functions should accept a single paramter: the object to serialize, and the JSON string, respectively. To modify the parameters of the stdlib JSON functions, you can use `functools.partial`:

```
# Do not escape unicode code-points.
my_json_dumps = functools.partial(json.dumps, ensure_ascii=False)

class SomeModel(Model):
    # Specify our custom serialization function.
    json_data = JSONField(json_dumps=my_json_dumps)
```

Let's look at some examples of using the SQLite json1 extension with Peewee. Here we'll prepare a database and a simple model for testing the json1 extension:

```
>>> from playhouse.sqlite_ext import *
>>> db = SqliteExtDatabase(':memory:')
>>> class KV(Model):
...     key = TextField()
...     value = JSONField()
...     class Meta:
...         database = db
...

>>> KV.create_table()
```

Storing data works as you might expect. There's no need to serialize dictionaries or lists as JSON, as this is done automatically by Peewee:

```
>>> KV.create(key='a', value={'k1': 'v1'})
<KV: 1>
>>> KV.get(KV.key == 'a').value
{'k1': 'v1'}
```

We can access specific parts of the JSON data using dictionary lookups:

```
>>> KV.get(KV.value['k1'] == 'v1').key
'a'
```

It's possible to update a JSON value in-place using the *update()* method. Note that "k1=v1" is preserved:

```
>>> KV.update(value=KV.value.update({'k2': 'v2', 'k3': 'v3'})).execute()
1
>>> KV.get(KV.key == 'a').value
{'k1': 'v1', 'k2': 'v2', 'k3': 'v3'}
```

We can also update existing data atomically, or remove keys by setting their value to `None`. In the following example, we'll update the value of "k1" and remove "k3" ("k2" will not be modified):

```
>>> KV.update(value=KV.value.update({'k1': 'v1-x', 'k3': None})).execute()
1
>>> KV.get(KV.key == 'a').value
{'k1': 'v1-x', 'k2': 'v2'}
```

We can also set individual parts of the JSON data using the *set()* method:

```
>>> KV.update(value=KV.value['k1'].set('v1')).execute()
1
>>> KV.get(KV.key == 'a').value
{'k1': 'v1', 'k2': 'v2'}
```

The *set()* method can also be used with objects, in addition to scalar values:

```
>>> KV.update(value=KV.value['k2'].set({'x2': 'y2'})).execute()
1
>>> KV.get(KV.key == 'a').value
{'k1': 'v1', 'k2': {'x2': 'y2'}}
```

Individual parts of the JSON data can be removed atomically as well, using *remove()*:

```
>>> KV.update(value=KV.value['k2'].remove()).execute()
1
>>> KV.get(KV.key == 'a').value
{'k1': 'v1'}
```

We can also get the type of value stored at a specific location in the JSON data using the *json_type()* method:

```
>>> KV.select(KV.value.json_type(), KV.value['k1'].json_type()).tuples()[:]
[('object', 'text')]
```

Let's add a nested value and then see how to iterate through it's contents recursively using the *tree()* method:

```
>>> KV.create(key='b', value={'x1': {'y1': 'z1', 'y2': 'z2'}, 'x2': [1, 2]})
<KV: 2>
>>> tree = KV.value.tree().alias('tree')
>>> query = KV.select(KV.key, tree.c.fullkey, tree.c.value).from_(KV, tree)
>>> query.tuples()[:]
[('a', '$', {'k1': 'v1'}),
 ('a', '$.k1', 'v1'),
 ('b', '$', {'x1': {'y1': 'z1', 'y2': 'z2'}, 'x2': [1, 2]}),
 ('b', '$.x2', [1, 2]),
 ('b', '$.x2[0]', 1),
```

(continues on next page)

(continued from previous page)

```
 ('b', '$.x2[1]', 2),
 ('b', '$.x1', {'y1': 'z1', 'y2': 'z2'}),
 ('b', '$.x1.y1', 'z1'),
 ('b', '$.x1.y2', 'z2')]
```

The *tree()* and *children()* methods are powerful. For more information on how to utilize them, see the json1 extension documentation.

Also note, that *JSONField* lookups can be chained:

```
>>> query = KV.select().where(KV.value['x1']['y1'] == 'z1')
>>> for obj in query:
...     print(obj.key, obj.value)
...

'b', {'x1': {'y1': 'z1', 'y2': 'z2'}, 'x2': [1, 2]}
```

For more information, refer to the sqlite json1 documentation.

__getitem__(*item*)

Parameters **item** – Access a specific key or array index in the JSON data.

Returns a special object exposing access to the JSON data.

Return type *JSONPath*

Access a specific key or array index in the JSON data. Returns a *JSONPath* object, which exposes convenient methods for reading or modifying a particular part of a JSON object.

Example:

```
# If metadata contains {"tags": ["list", "of", "tags"]}, we can
# extract the first tag in this way:
Post.select(Post, Post.metadata['tags'][0].alias('first_tag'))
```

For more examples see the *JSONPath* API documentation.

set(*value*[, *as_json=None*])

Parameters

- **value** – a scalar value, list, or dictionary.
- **as_json** (*bool*) – force the value to be treated as JSON, in which case it will be serialized as JSON in Python beforehand. By default, lists and dictionaries are treated as JSON to be serialized, while strings and integers are passed as-is.

Set the value stored in a *JSONField*.

Uses the json_set() function from the json1 extension.

update(*data*)

Parameters **data** – a scalar value, list or dictionary to merge with the data currently stored in a *JSONField*. To remove a particular key, set that key to `None` in the updated data.

Merge new data into the JSON value using the RFC-7396 MergePatch algorithm to apply a patch (`data` parameter) against the column data. MergePatch can add, modify, or delete elements of a JSON object, which means *update()* is a generalized replacement for both *set()* and *remove()*. MergePatch treats JSON array objects as atomic, so `update()` cannot append to an array, nor modify individual elements of an array.

For more information as well as examples, see the SQLite json_patch() function documentation.

remove()
Remove the data stored in the *JSONField*.

Uses the json_remove function from the json1 extension.

json_type()
Return a string identifying the type of value stored in the column.

The type returned will be one of:

- object
- array
- integer
- real
- true
- false
- text
- null <– the string "null" means an actual NULL value
- NULL <– an actual NULL value means the path was not found

Uses the json_type function from the json1 extension.

length()
Return the length of the array stored in the column.

Uses the json_array_length function from the json1 extension.

children()
The `children` function corresponds to `json_each`, a table-valued function that walks the JSON value provided and returns the immediate children of the top-level array or object. If a path is specified, then that path is treated as the top-most element.

The rows returned by calls to `children()` have the following attributes:

- `key`: the key of the current element relative to its parent.
- `value`: the value of the current element.
- `type`: one of the data-types (see *json_type()*).
- `atom`: the scalar value for primitive types, `NULL` for arrays and objects.
- `id`: a unique ID referencing the current node in the tree.
- `parent`: the ID of the containing node.
- `fullkey`: the full path describing the current element.
- `path`: the path to the container of the current row.

Internally this method uses the json_each (documentation link) function from the json1 extension.

Example usage (compare to *tree()* method):

```
class KeyData(Model):
    key = TextField()
    data = JSONField()
```

(continues on next page)

(continued from previous page)

```
KeyData.create(key='a', data={'k1': 'v1', 'x1': {'y1': 'z1'}})
KeyData.create(key='b', data={'x1': {'y1': 'z1', 'y2': 'z2'}})

# We will query the KeyData model for the key and all the
# top-level keys and values in it's data field.
kd = KeyData.data.children().alias('children')
query = (KeyData
         .select(kd.c.key, kd.c.value, kd.c.fullkey)
         .from_(KeyData, kd)
         .order_by(kd.c.key)
         .tuples())
print(query[:])

# PRINTS:
[('a', 'k1', 'v1',                    '$.k1'),
 ('a', 'x1', '{"y1":"z1"}',           '$.x1'),
 ('b', 'x1', '{"y1":"z1","y2":"z2"}', '$.x1')]
```

tree()

The `tree` function corresponds to `json_tree`, a table-valued function that recursively walks the JSON value provided and returns information about the keys at each level. If a path is specified, then that path is treated as the top-most element.

The rows returned by calls to `tree()` have the same attributes as rows returned by calls to *children()*:

- `key`: the key of the current element relative to its parent.
- `value`: the value of the current element.
- `type`: one of the data-types (see *json_type()*).
- `atom`: the scalar value for primitive types, `NULL` for arrays and objects.
- `id`: a unique ID referencing the current node in the tree.
- `parent`: the ID of the containing node.
- `fullkey`: the full path describing the current element.
- `path`: the path to the container of the current row.

Internally this method uses the json_tree (documentation link) function from the json1 extension.

Example usage:

```
class KeyData(Model):
    key = TextField()
    data = JSONField()

KeyData.create(key='a', data={'k1': 'v1', 'x1': {'y1': 'z1'}})
KeyData.create(key='b', data={'x1': {'y1': 'z1', 'y2': 'z2'}})

# We will query the KeyData model for the key and all the
# keys and values in it's data field, recursively.
kd = KeyData.data.tree().alias('tree')
query = (KeyData
         .select(kd.c.key, kd.c.value, kd.c.fullkey)
         .from_(KeyData, kd)
         .order_by(kd.c.key)
         .tuples())
```

(continues on next page)

(continued from previous page)

```
print(query[:])

# PRINTS:
[('a',  None,  '{"k1":"v1","x1":{"y1":"z1"}}', '$'),
 ('b',  None,  '{"x1":{"y1":"z1","y2":"z2"}}', '$'),
 ('a',  'k1',  'v1',                           '$.k1'),
 ('a',  'x1',  '{"y1":"z1"}',                  '$.x1'),
 ('b',  'x1',  '{"y1":"z1","y2":"z2"}',        '$.x1'),
 ('a',  'y1',  'z1',                           '$.x1.y1'),
 ('b',  'y1',  'z1',                           '$.x1.y1'),
 ('b',  'y2',  'z2',                           '$.x1.y2')]
```

class JSONPath(*field*[, *path=None*])

Parameters

- **field** (JSONField) – the field object we intend to access.
- **path** (*tuple*) – Components comprising the JSON path.

A convenient, Pythonic way of representing JSON paths for use with *JSONField*.

The JSONPath object implements __getitem__, accumulating path components, which it can turn into the corresponding json-path expression.

__getitem__(*item*)

Parameters **item** – Access a sub-key key or array index.

Returns a *JSONPath* representing the new path.

Access a sub-key or array index in the JSON data. Returns a *JSONPath* object, which exposes convenient methods for reading or modifying a particular part of a JSON object.

Example:

```
# If metadata contains {"tags": ["list", "of", "tags"]}, we can
# extract the first tag in this way:
first_tag = Post.metadata['tags'][0]
query = (Post
         .select(Post, first_tag.alias('first_tag'))
         .order_by(first_tag))
```

set(*value*[, *as_json=None*])

Parameters

- **value** – a scalar value, list, or dictionary.
- **as_json** (*bool*) – force the value to be treated as JSON, in which case it will be serialized as JSON in Python beforehand. By default, lists and dictionaries are treated as JSON to be serialized, while strings and integers are passed as-is.

Set the value at the given location in the JSON data.

Uses the json_set() function from the json1 extension.

update(*data*)

Parameters **data** – a scalar value, list or dictionary to merge with the data at the given location in the JSON data. To remove a particular key, set that key to None in the updated data.

Merge new data into the JSON value using the RFC-7396 MergePatch algorithm to apply a patch (`data` parameter) against the column data. MergePatch can add, modify, or delete elements of a JSON object, which means *`update()`* is a generalized replacement for both *`set()`* and *`remove()`*. MergePatch treats JSON array objects as atomic, so `update()` cannot append to an array, nor modify individual elements of an array.

For more information as well as examples, see the SQLite json_patch() function documentation.

remove()
Remove the data stored in at the given location in the JSON data.

Uses the json_type function from the json1 extension.

json_type()
Return a string identifying the type of value stored at the given location in the JSON data.

The type returned will be one of:

- object
- array
- integer
- real
- true
- false
- text
- null <– the string "null" means an actual NULL value
- NULL <– an actual NULL value means the path was not found

Uses the json_type function from the json1 extension.

length()
Return the length of the array stored at the given location in the JSON data.

Uses the json_array_length function from the json1 extension.

children()
Table-valued function that exposes the direct descendants of a JSON object at the given location. See also *`JSONField.children()`*.

tree()
Table-valued function that exposes all descendants, recursively, of a JSON object at the given location. See also *`JSONField.tree()`*.

class SearchField([*unindexed=False*[, *column_name=None*]])
Field-class to be used for columns on models representing full-text search virtual tables. The full-text search extensions prohibit the specification of any typing or constraints on columns. This behavior is enforced by the *`SearchField`*, which raises an exception if any configuration is attempted that would be incompatible with the full-text search extensions.

Example model for document search index (timestamp is stored in the table but it's data is not searchable):

```
class DocumentIndex(FTSModel):
    title = SearchField()
    content = SearchField()
    tags = SearchField()
    timestamp = SearchField(unindexed=True)
```

match(*term*)

Parameters **term** (*str*) – full-text search query/terms

Returns a *Expression* corresponding to the MATCH operator.

Sqlite's full-text search supports searching either the full table, including all indexed columns, **or** searching individual columns. The *match()* method can be used to restrict search to a single column:

```
class SearchIndex(FTSModel):
    title = SearchField()
    body = SearchField()

# Search *only* the title field and return results ordered by
# relevance, using bm25.
query = (SearchIndex
         .select(SearchIndex, SearchIndex.bm25().alias('score'))
         .where(SearchIndex.title.match('python'))
         .order_by(SearchIndex.bm25()))
```

To instead search *all* indexed columns, use the *FTSModel.match()* method:

```
# Searches *both* the title and body and return results ordered by
# relevance, using bm25.
query = (SearchIndex
         .select(SearchIndex, SearchIndex.bm25().alias('score'))
         .where(SearchIndex.match('python'))
         .order_by(SearchIndex.bm25()))
```

class VirtualModel

Model class designed to be used to represent virtual tables. The default metadata settings are slightly different, to match those frequently used by virtual tables.

Metadata options:

- arguments - arguments passed to the virtual table constructor.
- extension_module - name of extension to use for virtual table.
- **options - a dictionary of settings to apply in virtual table** constructor.
- primary_key - defaults to False, indicating no primary key.

These all are combined in the following way:

```
CREATE VIRTUAL TABLE <table_name>
USING <extension_module>
([prefix_arguments, ...] fields, ... [arguments, ...], [options...])
```

class FTSModel

Subclass of *VirtualModel* to be used with the FTS3 and FTS4 full-text search extensions.

FTSModel subclasses should be defined normally, however there are a couple caveats:

- Unique constraints, not null constraints, check constraints and foreign keys are not supported.
- Indexes on fields and multi-column indexes are ignored completely
- Sqlite will treat all column types as TEXT (although you can store other data types, Sqlite will treat them as text).
- FTS models contain a rowid field which is automatically created and managed by SQLite (unless you choose to explicitly set it during model creation). Lookups on this column **are fast and efficient**.

Given these constraints, it is strongly recommended that all fields declared on an `FTSModel` subclass be instances of *SearchField* (though an exception is made for explicitly declaring a *RowIDField*). Using *SearchField* will help prevent you accidentally creating invalid column constraints. If you wish to store metadata in the index but would not like it to be included in the full-text index, then specify `unindexed=True` when instantiating the *SearchField*.

The only exception to the above is for the `rowid` primary key, which can be declared using *RowIDField*. Lookups on the `rowid` are very efficient. If you are using FTS4 you can also use *DocIDField*, which is an alias for the rowid (though there is no benefit to doing so).

Because of the lack of secondary indexes, it usually makes sense to use the `rowid` primary key as a pointer to a row in a regular table. For example:

```
class Document(Model):
    # Canonical source of data, stored in a regular table.
    author = ForeignKeyField(User, backref='documents')
    title = TextField(null=False, unique=True)
    content = TextField(null=False)
    timestamp = DateTimeField()

    class Meta:
        database = db

class DocumentIndex(FTSModel):
    # Full-text search index.
    rowid = RowIDField()
    title = SearchField()
    content = SearchField()

    class Meta:
        database = db
        # Use the porter stemming algorithm to tokenize content.
        options = {'tokenize': 'porter'}
```

To store a document in the document index, we will `INSERT` a row into the `DocumentIndex` table, manually setting the `rowid` so that it matches the primary-key of the corresponding `Document`:

```
def store_document(document):
    DocumentIndex.insert({
        DocumentIndex.rowid: document.id,
        DocumentIndex.title: document.title,
        DocumentIndex.content: document.content}).execute()
```

To perform a search and return ranked results, we can query the `Document` table and join on the `DocumentIndex`. This join will be efficient because lookups on an FTSModel's `rowid` field are fast:

```
def search(phrase):
    # Query the search index and join the corresponding Document
    # object on each search result.
    return (Document
            .select()
            .join(
                DocumentIndex,
                on=(Document.id == DocumentIndex.rowid))
            .where(DocumentIndex.match(phrase))
            .order_by(DocumentIndex.bm25()))
```

> **Warning:** All SQL queries on `FTSModel` classes will be full-table scans **except** full-text searches and `rowid` lookups.

If the primary source of the content you are indexing exists in a separate table, you can save some disk space by instructing SQLite to not store an additional copy of the search index content. SQLite will still create the metadata and data-structures needed to perform searches on the content, but the content itself will not be stored in the search index.

To accomplish this, you can specify a table or column using the `content` option. The FTS4 documentation has more information.

Here is a short example illustrating how to implement this with peewee:

```
class Blog(Model):
    title = TextField()
    pub_date = DateTimeField(default=datetime.datetime.now)
    content = TextField()  # We want to search this.

    class Meta:
        database = db

class BlogIndex(FTSModel):
    content = SearchField()

    class Meta:
        database = db
        options = {'content': Blog.content}  # <-- specify data source.

db.create_tables([Blog, BlogIndex])

# Now, we can manage content in the BlogIndex. To populate the
# search index:
BlogIndex.rebuild()

# Optimize the index.
BlogIndex.optimize()
```

The `content` option accepts either a single *Field* or a *Model* and can reduce the amount of storage used by the database file. However, content will need to be manually moved to/from the associated `FTSModel`.

classmethod match(*term*)

Parameters **term** – Search term or expression.

Generate a SQL expression representing a search for the given term or expression in the table. SQLite uses the `MATCH` operator to indicate a full-text search.

Example:

```
# Search index for "search phrase" and return results ranked
# by relevancy using the BM25 algorithm.
query = (DocumentIndex
         .select()
         .where(DocumentIndex.match('search phrase'))
         .order_by(DocumentIndex.bm25()))
for result in query:
    print('Result: %s' % result.title)
```

classmethod search(*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*[, *explicit_ordering=False*]]]])

Parameters

- **term** (*str*) – Search term to use.
- **weights** – A list of weights for the columns, ordered with respect to the column's position in the table. **Or**, a dictionary keyed by the field or field name and mapped to a value.
- **with_score** – Whether the score should be returned as part of the `SELECT` statement.
- **score_alias** (*str*) – Alias to use for the calculated rank score. This is the attribute you will use to access the score if `with_score=True`.
- **explicit_ordering** (*bool*) – Order using full SQL function to calculate rank, as opposed to simply referencing the score alias in the ORDER BY clause.

Shorthand way of searching for a term and sorting results by the quality of the match.

Note: This method uses a simplified algorithm for determining the relevance rank of results. For more sophisticated result ranking, use the *search_bm25()* method.

```
# Simple search.
docs = DocumentIndex.search('search term')
for result in docs:
    print(result.title)

# More complete example.
docs = DocumentIndex.search(
    'search term',
    weights={'title': 2.0, 'content': 1.0},
    with_score=True,
    score_alias='search_score')
for result in docs:
    print(result.title, result.search_score)
```

classmethod search_bm25(*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*[, *explicit_ordering=False*]]]])

Parameters

- **term** (*str*) – Search term to use.
- **weights** – A list of weights for the columns, ordered with respect to the column's position in the table. **Or**, a dictionary keyed by the field or field name and mapped to a value.
- **with_score** – Whether the score should be returned as part of the `SELECT` statement.
- **score_alias** (*str*) – Alias to use for the calculated rank score. This is the attribute you will use to access the score if `with_score=True`.
- **explicit_ordering** (*bool*) – Order using full SQL function to calculate rank, as opposed to simply referencing the score alias in the ORDER BY clause.

Shorthand way of searching for a term and sorting results by the quality of the match using the BM25 algorithm.

> **Attention:** The BM25 ranking algorithm is only available for FTS4. If you are using FTS3, use the *search()* method instead.

classmethod search_bm25f(*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*[, *explicit_ordering=False*]]]])

Same as *FTSModel.search_bm25()*, but using the BM25f variant of the BM25 ranking algorithm.

classmethod search_lucene(*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*[, *explicit_ordering=False*]]]])

Same as *FTSModel.search_bm25()*, but using the result ranking algorithm from the Lucene search engine.

classmethod rank([*col1_weight*, *col2_weight...coln_weight*])

Parameters **col_weight** (*float*) – (Optional) weight to give to the *ith* column of the model. By default all columns have a weight of `1.0`.

Generate an expression that will calculate and return the quality of the search match. This `rank` can be used to sort the search results. A higher rank score indicates a better match.

The `rank` function accepts optional parameters that allow you to specify weights for the various columns. If no weights are specified, all columns are considered of equal importance.

Note: The algorithm used by *rank()* is simple and relatively quick. For more sophisticated result ranking, use:

- *bm25()*
- *bm25f()*
- *lucene()*

```
query = (DocumentIndex
         .select(
             DocumentIndex,
             DocumentIndex.rank().alias('score'))
         .where(DocumentIndex.match('search phrase'))
         .order_by(DocumentIndex.rank()))

for search_result in query:
    print search_result.title, search_result.score
```

classmethod bm25([*col1_weight*, *col2_weight...coln_weight*])

Parameters **col_weight** (*float*) – (Optional) weight to give to the *ith* column of the model. By default all columns have a weight of `1.0`.

Generate an expression that will calculate and return the quality of the search match using the BM25 algorithm. This value can be used to sort the search results, with higher scores corresponding to better matches.

Like *rank()*, `bm25` function accepts optional parameters that allow you to specify weights for the various columns. If no weights are specified, all columns are considered of equal importance.

> **Attention:** The BM25 result ranking algorithm requires FTS4. If you are using FTS3, use *rank()* instead.

```
query = (DocumentIndex
         .select(
             DocumentIndex,
             DocumentIndex.bm25().alias('score'))
         .where(DocumentIndex.match('search phrase'))
         .order_by(DocumentIndex.bm25()))

for search_result in query:
    print(search_result.title, search_result.score)
```

Note: The above code example is equivalent to calling the *search_bm25()* method:

```
query = DocumentIndex.search_bm25('search phrase', with_score=True)
for search_result in query:
    print(search_result.title, search_result.score)
```

classmethod bm25f([*col1_weight*, *col2_weight...coln_weight*])
Identical to *bm25()*, except that it uses the BM25f variant of the BM25 ranking algorithm.

classmethod lucene([*col1_weight*, *col2_weight...coln_weight*])
Identical to *bm25()*, except that it uses the Lucene search result ranking algorithm.

classmethod rebuild()
Rebuild the search index – this only works when the `content` option was specified during table creation.

classmethod optimize()
Optimize the search index.

class FTS5Model
Subclass of *VirtualModel* to be used with the FTS5 full-text search extensions.

FTS5Model subclasses should be defined normally, however there are a couple caveats:

- FTS5 explicitly disallows specification of any constraints, data-type or indexes on columns. For that reason, all columns **must** be instances of *SearchField*.
- FTS5 models contain a `rowid` field which is automatically created and managed by SQLite (unless you choose to explicitly set it during model creation). Lookups on this column **are fast and efficient**.
- Indexes on fields and multi-column indexes are not supported.

The `FTS5` extension comes with a built-in implementation of the BM25 ranking function. Therefore, the `search` and `search_bm25` methods have been overridden to use the builtin ranking functions rather than user-defined functions.

classmethod fts5_installed()
Return a boolean indicating whether the FTS5 extension is installed. If it is not installed, an attempt will be made to load the extension.

classmethod search(*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*]]])

Parameters

- **term** (*str*) – Search term to use.
- **weights** – A list of weights for the columns, ordered with respect to the column's position in the table. **Or**, a dictionary keyed by the field or field name and mapped to a value.
- **with_score** – Whether the score should be returned as part of the `SELECT` statement.

- **score_alias** (*str*) – Alias to use for the calculated rank score. This is the attribute you will use to access the score if `with_score=True`.
- **explicit_ordering** (*bool*) – Order using full SQL function to calculate rank, as opposed to simply referencing the score alias in the ORDER BY clause.

Shorthand way of searching for a term and sorting results by the quality of the match. The `FTS5` extension provides a built-in implementation of the BM25 algorithm, which is used to rank the results by relevance.

Higher scores correspond to better matches.

```
# Simple search.
docs = DocumentIndex.search('search term')
for result in docs:
    print(result.title)

# More complete example.
docs = DocumentIndex.search(
    'search term',
    weights={'title': 2.0, 'content': 1.0},
    with_score=True,
    score_alias='search_score')
for result in docs:
    print(result.title, result.search_score)
```

classmethod search_bm25(*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*]]])

With FTS5, *search_bm25()* is identical to the *search()* method.

classmethod rank([*col1_weight*, *col2_weight...coln_weight*])

Parameters **col_weight** (*float*) – (Optional) weight to give to the *ith* column of the model. By default all columns have a weight of `1.0`.

Generate an expression that will calculate and return the quality of the search match using the BM25 algorithm. This value can be used to sort the search results, with higher scores corresponding to better matches.

The *rank()* function accepts optional parameters that allow you to specify weights for the various columns. If no weights are specified, all columns are considered of equal importance.

```
query = (DocumentIndex
         .select(
             DocumentIndex,
             DocumentIndex.rank().alias('score'))
         .where(DocumentIndex.match('search phrase'))
         .order_by(DocumentIndex.rank()))

for search_result in query:
    print(search_result.title, search_result.score)
```

Note: The above code example is equivalent to calling the *search()* method:

```
query = DocumentIndex.search('search phrase', with_score=True)
for search_result in query:
    print(search_result.title, search_result.score)
```

---

classmethod bm25([*col1_weight*, *col2_weight...coln_weight*])

Because FTS5 provides built-in support for BM25, the *bm25()* method is identical to the *rank()* method.

classmethod VocabModel([*table_type='row'|'col'|'instance'*[, *table_name=None*]])

Parameters

- **table_type** (*str*) – Either 'row', 'col' or 'instance'.
- **table_name** – Name for the vocab table. If not specified, will be "fts5tablename_v".

Generate a model class suitable for accessing the vocab table corresponding to FTS5 search index.

class TableFunction

Implement a user-defined table-valued function. Unlike a simple *scalar or aggregate* function, which returns a single scalar value, a table-valued function can return any number of rows of tabular data.

Simple example:

```
from playhouse.sqlite_ext import TableFunction


class Series(TableFunction):
    # Name of columns in each row of generated data.
    columns = ['value']

    # Name of parameters the function may be called with.
    params = ['start', 'stop', 'step']

    def initialize(self, start=0, stop=None, step=1):
        """
        Table-functions declare an initialize() method, which is
        called with whatever arguments the user has called the
        function with.
        """
        self.start = self.current = start
        self.stop = stop or float('Inf')
        self.step = step

    def iterate(self, idx):
        """
        Iterate is called repeatedly by the SQLite database engine
        until the required number of rows has been read **or** the
        function raises a `StopIteration` signalling no more rows
        are available.
        """
        if self.current > self.stop:
            raise StopIteration

        ret, self.current = self.current, self.current + self.step
        return (ret,)

# Register the table-function with our database, which ensures it
# is declared whenever a connection is opened.
db.table_function('series')(Series)

# Usage:
cursor = db.execute_sql('SELECT * FROM series(?, ?, ?)', (0, 5, 2))
for value, in cursor:
    print(value)
```

Note: A *TableFunction* must be registered with a database connection before it can be used. To ensure the table function is always available, you can use the *SqliteDatabase.table_function()* decorator to register the function with the database.

TableFunction implementations must provide two attributes and implement two methods, described below.

columns
A list containing the names of the columns for the data returned by the function. For example, a function that is used to split a string on a delimiter might specify 3 columns: `[substring, start_idx, end_idx]`.

params
The names of the parameters the function may be called with. All parameters, including optional parameters, should be listed. For example, a function that is used to split a string on a delimiter might specify 2 params: `[string, delimiter]`.

name
Optional - specify the name for the table function. If not provided, name will be taken from the class name.

print_tracebacks = True
Print a full traceback for any errors that occur in the table-function's callback methods. When set to False, only the generic OperationalError will be visible.

initialize(***parameter_values*)

Parameters **parameter_values** – Parameters the function was called with.

Returns No return value.

The `initialize` method is called to initialize the table function with the parameters the user specified when calling the function.

iterate(*idx*)

Parameters **idx** (*int*) – current iteration step

Returns A tuple of row data corresponding to the columns named in the *columns* attribute.

Raises **StopIteration** – To signal that no more rows are available.

This function is called repeatedly and returns successive rows of data. The function may terminate before all rows are consumed (especially if the user specified a `LIMIT` on the results). Alternatively, the function can signal that no more data is available by raising a `StopIteration` exception.

classmethod register(*conn*)

Parameters **conn** – A `sqlite3.Connection` object.

Register the table function with a DB-API 2.0 `sqlite3.Connection` object. Table-valued functions **must** be registered before they can be used in a query.

Example:

```
class MyTableFunction(TableFunction):
    name = 'my_func'
    # ... other attributes and methods ...

db = SqliteDatabase(':memory:')
db.connect()
```

(continues on next page)

(continued from previous page)

```
MyTableFunction.register(db.connection())
```

To ensure the *TableFunction* is registered every time a connection is opened, use the *table_function()* decorator.

ClosureTable(*model_class*[, *foreign_key=None*[, *referencing_class=None*[, *referencing_key=None*]]])

Parameters

- **model_class** – The model class containing the nodes in the tree.
- **foreign_key** – The self-referential parent-node field on the model class. If not provided, peewee will introspect the model to find a suitable key.
- **referencing_class** – Intermediate table for a many-to-many relationship.
- **referencing_key** – For a many-to-many relationship, the originating side of the relation.

Returns Returns a *VirtualModel* for working with a closure table.

Factory function for creating a model class suitable for working with a transitive closure table. Closure tables are *VirtualModel* subclasses that work with the transitive closure SQLite extension. These special tables are designed to make it easy to efficiently query hierarchical data. The SQLite extension manages an AVL tree behind-the-scenes, transparently updating the tree when your table changes and making it easy to perform common queries on hierarchical data.

To use the closure table extension in your project, you need:

1. A copy of the SQLite extension. The source code can be found in the SQLite code repository or by cloning this gist:

```
$ git clone https://gist.github.com/coleifer/7f3593c5c2a645913b92 closure
$ cd closure/
```

2. Compile the extension as a shared library, e.g.

```
$ gcc -g -fPIC -shared closure.c -o closure.so
```

3. Create a model for your hierarchical data. The only requirement here is that the model has an integer primary key and a self-referential foreign key. Any additional fields are fine.

```
class Category(Model):
    name = CharField()
    metadata = TextField()
    parent = ForeignKeyField('self', index=True, null=True)  # Required.

# Generate a model for the closure virtual table.
CategoryClosure = ClosureTable(Category)
```

The self-referentiality can also be achieved via an intermediate table (for a many-to-many relation).

```
class User(Model):
    name = CharField()

class UserRelations(Model):
    user = ForeignKeyField(User)
    knows = ForeignKeyField(User, backref='_known_by')
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        primary_key = CompositeKey('user', 'knows') # Alternatively, a unique␣
↪index on both columns.

# Generate a model for the closure virtual table, specifying the␣
↪UserRelations as the referencing table
UserClosure = ClosureTable(
    User,
    referencing_class=UserRelations,
    foreign_key=UserRelations.knows,
    referencing_key=UserRelations.user)
```

4. In your application code, make sure you load the extension when you instantiate your *Database* object. This is done by passing the path to the shared library to the `load_extension()` method.

```
db = SqliteExtDatabase('my_database.db')
db.load_extension('/path/to/closure')
```

Warning: There are two caveats you should be aware of when using the `transitive_closure` extension. First, it requires that your *source model* have an integer primary key. Second, it is strongly recommended that you create an index on the self-referential foreign key.

Example:

```
class Category(Model):
    name = CharField()
    metadata = TextField()
    parent = ForeignKeyField('self', index=True, null=True)  # Required.

# Generate a model for the closure virtual table.
CategoryClosure = ClosureTable(Category)

 # Create the tables if they do not exist.
 db.create_tables([Category, CategoryClosure], True)
```

It is now possible to perform interesting queries using the data from the closure table:

```
# Get all ancestors for a particular node.
laptops = Category.get(Category.name == 'Laptops')
for parent in Closure.ancestors(laptops):
    print parent.name

# Computer Hardware
# Computers
# Electronics
# All products

# Get all descendants for a particular node.
hardware = Category.get(Category.name == 'Computer Hardware')
for node in Closure.descendants(hardware):
    print node.name

# Laptops
```

(continues on next page)

(continued from previous page)

```
# Desktops
# Hard-drives
# Monitors
# LCD Monitors
# LED Monitors
```

API of the *VirtualModel* returned by *ClosureTable()*.

class BaseClosureTable

id
A field for the primary key of the given node.

depth
A field representing the relative depth of the given node.

root
A field representing the relative root node.

descendants(*node*[, *depth=None*[, *include_node=False*]])
Retrieve all descendants of the given node. If a depth is specified, only nodes at that depth (relative to the given node) will be returned.

```
node = Category.get(Category.name == 'Electronics')

# Direct child categories.
children = CategoryClosure.descendants(node, depth=1)

# Grand-child categories.
children = CategoryClosure.descendants(node, depth=2)

# Descendants at all depths.
all_descendants = CategoryClosure.descendants(node)
```

ancestors(*node*[, *depth=None*[, *include_node=False*]])
Retrieve all ancestors of the given node. If a depth is specified, only nodes at that depth (relative to the given node) will be returned.

```
node = Category.get(Category.name == 'Laptops')

# All ancestors.
all_ancestors = CategoryClosure.ancestors(node)

# Grand-parent category.
grandparent = CategoryClosure.ancestores(node, depth=2)
```

siblings(*node*[, *include_node=False*])
Retrieve all nodes that are children of the specified node's parent.

Note: For an in-depth discussion of the SQLite transitive closure extension, check out this blog post, Querying Tree Structures in SQLite using Python and the Transitive Closure Extension.

---

class LSMTable
VirtualModel subclass suitable for working with the lsm1 extension The *lsm1* extension is a virtual table that provides a SQL interface to the lsm key/value storage engine from SQLite4.

Note: The LSM1 extension has not been released yet (SQLite version 3.22 at time of writing), so consider this feature experimental with potential to change in subsequent releases.

LSM tables define one primary key column and an arbitrary number of additional value columns (which are serialized and stored in a single value field in the storage engine). The primary key must be all of the same type and use one of the following field types:

- *IntegerField*
- *TextField*
- *BlobField*

Since the LSM storage engine is a key/value store, primary keys (including integers) must be specified by the application.

Attention: Secondary indexes are not supported by the LSM engine, so the only efficient queries will be lookups (or range queries) on the primary key. Other fields can be queried and filtered on, but may result in a full table-scan.

Example model declaration:

```
db = SqliteExtDatabase('my_app.db')
db.load_extension('lsm.so')  # Load shared library.

class EventLog(LSMTable):
    timestamp = IntegerField(primary_key=True)
    action = TextField()
    sender = TextField()
    target = TextField()

    class Meta:
        database = db
        filename = 'eventlog.ldb'  # LSM data is stored in separate db.

# Declare virtual table.
EventLog.create_table()
```

Example queries:

```
# Use dictionary operators to get, set and delete rows from the LSM
# table. Slices may be passed to represent a range of key values.
def get_timestamp():
    # Return time as integer expressing time in microseconds.
    return int(time.time() * 1000000)

# Create a new row, at current timestamp.
ts = get_timestamp()
EventLog[ts] = ('pageview', 'search', '/blog/some-post/')

# Retrieve row from event log.
log = EventLog[ts]
print(log.action, log.sender, log.target)
# Prints ("pageview", "search", "/blog/some-post/")
```

(continues on next page)

(continued from previous page)

```
# Delete the row.
del EventLog[ts]

# We can also use the "create()" method.
EventLog.create(
    timestamp=get_timestamp(),
    action='signup',
    sender='newsletter',
    target='sqlite-news')
```

Simple key/value model declaration:

```
class KV(LSMTable):
    key = TextField(primary_key=True)
    value = TextField()

    class Meta:
        database = db
        filename = 'kv.ldb'

db.create_tables([KV])
```

For tables consisting of a single value field, Peewee will return the value directly when getting a single item. You can also request slices of rows, in which case Peewee returns a corresponding *Select* query, which can be iterated over. Below are some examples:

```
>>> KV['k0'] = 'v0'
>>> print(KV['k0'])
'v0'

>>> data = [{'key': 'k%d' % i, 'value': 'v%d' % i} for i in range(20)]
>>> KV.insert_many(data).execute()

>>> KV.select().count()
20

>>> KV['k8']
'v8'

>>> list(KV['k4.1':'k7.x']
[Row(key='k5', value='v5'),
 Row(key='k6', value='v6'),
 Row(key='k7', value='v7')]

>>> list(KV['k6xxx':])
[Row(key='k7', value='v7'),
 Row(key='k8', value='v8'),
 Row(key='k9', value='v9')]
```

You can also index the *LSMTable* using expressions:

```
>>> list(KV[KV.key > 'k6'])
[Row(key='k7', value='v7'),
 Row(key='k8', value='v8'),
 Row(key='k9', value='v9')]
```

(continues on next page)

(continued from previous page)

```
>>> list(KV[(KV.key > 'k6') & (KV.value != 'v8')])
[Row(key='k7', value='v7'),
 Row(key='k9', value='v9')]
```

You can delete single rows using `del` or multiple rows using slices or expressions:

```
>>> del KV['k1']
>>> del KV['k3x':'k8']
>>> del KV[KV.key.between('k10', 'k18')]

>>> list(KV[:])
[Row(key='k0', value='v0'),
 Row(key='k19', value='v19'),
 Row(key='k2', value='v2'),
 Row(key='k3', value='v3'),
 Row(key='k9', value='v9')]
```

Attempting to get a single non-existant key will result in a `KeyError`, but slices will not raise an exception:

```
>>> KV['k1']
...
KeyError: 'k1'

>>> list(KV['k1':'k1'])
[]
```

class ZeroBlob(*length*)

Parameters **length** (*int*) – Size of blob in bytes.

ZeroBlob is used solely to reserve space for storing a BLOB that supports incremental I/O. To use the SQLite BLOB-store it is necessary to first insert a ZeroBlob of the desired size into the row you wish to use with incremental I/O.

For example, see *Blob*.

class Blob(*database*, *table*, *column*, *rowid*[, *read_only=False*])

Parameters

- **database** – *SqliteExtDatabase* instance.
- **table** (*str*) – Name of table being accessed.
- **column** (*str*) – Name of column being accessed.
- **rowid** (*int*) – Primary-key of row being accessed.
- **read_only** (*bool*) – Prevent any modifications to the blob data.

Open a blob, stored in the given table/column/row, for incremental I/O. To allocate storage for new data, you can use the *ZeroBlob*, which is very efficient.

```
class RawData(Model):
    data = BlobField()

# Allocate 100MB of space for writing a large file incrementally:
query = RawData.insert({'data': ZeroBlob(1024 * 1024 * 100)})
rowid = query.execute()
```

(continues on next page)

(continued from previous page)

```
# Now we can open the row for incremental I/O:
blob = Blob(db, 'rawdata', 'data', rowid)

# Read from the file and write to the blob in chunks of 4096 bytes.
while True:
    data = file_handle.read(4096)
    if not data:
        break
    blob.write(data)

bytes_written = blob.tell()
blob.close()
```

read(*[n=None]*)

Parameters **n** (*int*) – Only read up to *n* bytes from current position in file.

Read up to *n* bytes from the current position in the blob file. If *n* is not specified, the entire blob will be read.

seek(*offset[, whence=0]*)

Parameters

- **offset** (*int*) – Seek to the given offset in the file.
- **whence** (*int*) – Seek relative to the specified frame of reference.

Values for `whence`:

- `0`: beginning of file
- `1`: current position
- `2`: end of file

tell()

Return current offset within the file.

write(*data*)

Parameters **data** (*bytes*) – Data to be written

Writes the given data, starting at the current position in the file.

close()

Close the file and free associated resources.

reopen(*rowid*)

Parameters **rowid** (*int*) – Primary key of row to open.

If a blob has already been opened for a given table/column, you can use the *reopen()* method to re-use the same *Blob* object for accessing multiple rows in the table.

1.12.3 Additional Features

The *SqliteExtDatabase* accepts an initialization option to register support for a simple bloom filter. The bloom filter, once initialized, can then be used for efficient membership queries on large set of data.

Here's an example:

```
db = CSqliteExtDatabase(':memory:', bloomfilter=True)

# Create and define a table to store some data.
db.execute_sql('CREATE TABLE "register" ("data" TEXT)')
Register = Table('register', ('data',)).bind(db)

# Populate the database with a bunch of text.
with db.atomic():
    for i in 'abcdefghijklmnopqrstuvwxyz':
        keys = [i * j for j in range(1, 10)]  # a, aa, aaa, ... aaaaaaaaa
        Register.insert([{'data': key} for key in keys]).execute()

# Collect data into a 16KB bloomfilter.
query = Register.select(fn.bloomfilter(Register.data, 16 * 1024).alias('buf'))
row = query.get()
buf = row['buf']

# Use bloomfilter buf to test whether other keys are members.
test_keys = (
    ('aaaa', True),
    ('abc', False),
    ('zzzzzzz', True),
    ('zyxwvut', False))
for key, is_present in test_keys:
    query = Register.select(fn.bloomfilter_contains(key, buf).alias('is_member'))
    answer = query.get()['is_member']
    assert answer == is_present
```

The *SqliteExtDatabase* can also register other useful functions:

- `rank_functions` (enabled by default): registers functions for ranking search results, such as *bm25* and *lucene*.
- `hash_functions`: registers md5, sha1, sha256, adler32, crc32 and murmurhash functions.
- `regexp_function`: registers a regexp function.

Examples:

```
def create_new_user(username, password):
    # DO NOT DO THIS IN REAL LIFE. PLEASE.
    query = User.insert({'username': username, 'password': fn.sha1(password)})
    new_user_id = query.execute()
```

You can use the *murmurhash* function to hash bytes to an integer for compact storage:

```
>>> db = SqliteExtDatabase(':memory:', hash_functions=True)
>>> db.execute_sql('SELECT murmurhash(?)', ('abcdefg',)).fetchone()
(4188131059,)
```

1.13 Playhouse, extensions to Peewee

Peewee comes with numerous extension modules which are collected under the `playhouse` namespace. Despite the silly name, there are some very useful extensions, particularly those that expose vendor-specific database features like the *SQLite Extensions* and *Postgresql Extensions* extensions.

Below you will find a loosely organized listing of the various modules that make up the `playhouse`.

Database drivers / vendor-specific database functionality

- *SQLite Extensions* (on its own page)
- *SqliteQ*
- *Sqlite User-Defined Functions*
- *apsw, an advanced sqlite driver*
- *Sqlcipher backend*
- *Postgresql Extensions*
- *Cockroach Database*
- *MySQL Extensions*

High-level features

- *Fields*
- *Shortcuts*
- *Hybrid Attributes*
- *Key/Value Store*
- *Signal support*
- *DataSet*

Database management and framework integration

- *pwiz, a model generator*
- *Schema Migrations*
- *Connection pool*
- *Reflection*
- *Database URL*
- *Test Utils*
- *Flask Utils*

1.13.1 Sqlite Extensions

The Sqlite extensions have been moved to *their own page*.

1.13.2 SqliteQ

The `playhouse.sqliteq` module provides a subclass of *`SqliteExtDatabase`*, that will serialize concurrent writes to a SQLite database. `SqliteQueueDatabase` can be used as a drop-in replacement for the regular *`SqliteDatabase`* if you want simple **read and write** access to a SQLite database from **multiple threads**.

SQLite only allows one connection to write to the database at any given time. As a result, if you have a multi-threaded application (like a web-server, for example) that needs to write to the database, you may see occasional errors when one or more of the threads attempting to write cannot acquire the lock.

`SqliteQueueDatabase` is designed to simplify things by sending all write queries through a single, long-lived connection. The benefit is that you get the appearance of multiple threads writing to the database without conflicts or

timeouts. The downside, however, is that you cannot issue write transactions that encompass multiple queries – all writes run in autocommit mode, essentially.

Note: The module gets its name from the fact that all write queries get put into a thread-safe queue. A single worker thread listens to the queue and executes all queries that are sent to it.

Transactions

Because all queries are serialized and executed by a single worker thread, it is possible for transactional SQL from separate threads to be executed out-of-order. In the example below, the transaction started by thread "B" is rolled back by thread "A" (with bad consequences!):

- Thread A: UPDATE transplants SET organ='liver', ...;
- Thread B: BEGIN TRANSACTION;
- Thread B: UPDATE life_support_system SET timer += 60 ...;
- Thread A: ROLLBACK; – Oh no....

Since there is a potential for queries from separate transactions to be interleaved, the `transaction()` and `atomic()` methods are disabled on `SqliteQueueDatabase`.

For cases when you wish to temporarily write to the database from a different thread, you can use the `pause()` and `unpause()` methods. These methods block the caller until the writer thread is finished with its current workload. The writer then disconnects and the caller takes over until `unpause` is called.

The `stop()`, `start()`, and `is_stopped()` methods can also be used to control the writer thread.

Note: Take a look at SQLite's isolation documentation for more information about how SQLite handles concurrent connections.

Code sample

Creating a database instance does not require any special handling. The `SqliteQueueDatabase` accepts some special parameters which you should be aware of, though. If you are using gevent, you must specify `use_gevent=True` when instantiating your database – this way Peewee will know to use the appropriate objects for handling queueing, thread creation, and locking.

```
from playhouse.sqliteq import SqliteQueueDatabase

db = SqliteQueueDatabase(
    'my_app.db',
    use_gevent=False,  # Use the standard library "threading" module.
    autostart=False,  # The worker thread now must be started manually.
    queue_max_size=64,  # Max. # of pending writes that can accumulate.
    results_timeout=5.0)  # Max. time to wait for query to be executed.
```

If `autostart=False`, as in the above example, you will need to call `start()` to bring up the worker threads that will do the actual write query execution.

```
@app.before_first_request
def _start_worker_threads():
    db.start()
```

If you plan on performing SELECT queries or generally wanting to access the database, you will need to call *connect()* and *close()* as you would with any other database instance.

When your application is ready to terminate, use the `stop()` method to shut down the worker thread. If there was a backlog of work, then this method will block until all pending work is finished (though no new work is allowed).

```
import atexit

@atexit.register
def _stop_worker_threads():
    db.stop()
```

Lastly, the `is_stopped()` method can be used to determine whether the database writer is up and running.

1.13.3 Sqlite User-Defined Functions

The `sqlite_udf` playhouse module contains a number of user-defined functions, aggregates, and table-valued functions, which you may find useful. The functions are grouped in collections and you can register these user-defined extensions individually, by collection, or register everything.

Scalar functions are functions which take a number of parameters and return a single value. For example, converting a string to upper-case, or calculating the MD5 hex digest.

Aggregate functions are like scalar functions that operate on multiple rows of data, producing a single result. For example, calculating the sum of a list of integers, or finding the smallest value in a particular column.

Table-valued functions are simply functions that can return multiple rows of data. For example, a regular-expression search function that returns all the matches in a given string, or a function that accepts two dates and generates all the intervening days.

Note: To use table-valued functions, you will need to build the `playhouse._sqlite_ext` C extension.

Registering user-defined functions:

```
db = SqliteDatabase('my_app.db')

# Register *all* functions.
register_all(db)

# Alternatively, you can register individual groups. This will just
# register the DATE and MATH groups of functions.
register_groups(db, 'DATE', 'MATH')

# If you only wish to register, say, the aggregate functions for a
# particular group or groups, you can:
register_aggregate_groups(db, 'DATE')

# If you only wish to register a single function, then you can:
from playhouse.sqlite_udf import gzip, gunzip
db.register_function(gzip, 'gzip')
db.register_function(gunzip, 'gunzip')
```

Using a library function ("hostname"):

```
# Assume we have a model, Link, that contains lots of arbitrary URLs.
# We want to discover the most common hosts that have been linked.
```

(continues on next page)

(continued from previous page)

```
query = (Link
         .select(fn.hostname(Link.url).alias('host'), fn.COUNT(Link.id))
         .group_by(fn.hostname(Link.url))
         .order_by(fn.COUNT(Link.id).desc())
         .tuples())

# Print the hostname along with number of links associated with it.
for host, count in query:
    print('%s: %s' % (host, count))
```

Functions, listed by collection name

Scalar functions are indicated by `(f)`, aggregate functions by `(a)`, and table-valued functions by `(t)`.

CONTROL_FLOW

if_then_else(*cond*, *truthy*[, *falsey=None*])

Simple ternary-type operator, where, depending on the truthiness of the `cond` parameter, either the `truthy` or `falsey` value will be returned.

DATE

strip_tz(*date_str*)

Parameters **date_str** – A datetime, encoded as a string.

Returns The datetime with any timezone info stripped off.

The time is not adjusted in any way, the timezone is simply removed.

humandelta(*nseconds*[, *glue=', '*])

Parameters

- **nseconds** (*int*) – Number of seconds, total, in timedelta.
- **glue** (*str*) – Fragment to join values.

Returns Easy-to-read description of timedelta.

Example, 86471 -> "1 day, 1 minute, 11 seconds"

mintdiff(*datetime_value*)

Parameters **datetime_value** – A date-time.

Returns Minimum difference between any two values in list.

Aggregate function that computes the minimum difference between any two datetimes.

avgtdiff(*datetime_value*)

Parameters **datetime_value** – A date-time.

Returns Average difference between values in list.

Aggregate function that computes the average difference between consecutive values in the list.

duration(*datetime_value*)

Parameters **datetime_value** – A date-time.

Returns Duration from smallest to largest value in list, in seconds.

Aggregate function that computes the duration from the smallest to the largest value in the list, returned in seconds.

date_series(*start*, *stop*[, *step_seconds=86400*])

Parameters

- **start** (*datetime*) – Start datetime
- **stop** (*datetime*) – Stop datetime
- **step_seconds** (*int*) – Number of seconds comprising a step.

Table-value function that returns rows consisting of the date/+time values encountered iterating from start to stop, `step_seconds` at a time.

Additionally, if start does not have a time component and step_seconds is greater-than-or-equal-to one day (86400 seconds), the values returned will be dates. Conversely, if start does not have a date component, values will be returned as times. Otherwise values are returned as datetimes.

Example:

```
SELECT * FROM date_series('2017-01-28', '2017-02-02');

value
-----
2017-01-28
2017-01-29
2017-01-30
2017-01-31
2017-02-01
2017-02-02
```

FILE

file_ext(*filename*)

Parameters **filename** (*str*) – Filename to extract extension from.

Returns Returns the file extension, including the leading ".".

file_read(*filename*)

Parameters **filename** (*str*) – Filename to read.

Returns Contents of the file.

HELPER

gzip(*data*[, *compression=9*])

Parameters

- **data** (*bytes*) – Data to compress.
- **compression** (*int*) – Compression level (9 is max).

Returns Compressed binary data.

gunzip(*data*)

Parameters **data** (*bytes*) – Compressed data.

Returns Uncompressed binary data.

hostname(*url*)

Parameters **url** (*str*) – URL to extract hostname from.

Returns hostname portion of URL

toggle(*key*)

Parameters **key** – Key to toggle.

Toggle a key between True/False state. Example:

```
>>> toggle('my-key')
True
>>> toggle('my-key')
False
>>> toggle('my-key')
True
```

setting(*key*[, *value=None*])

Parameters

- **key** – Key to set/retrieve.
- **value** – Value to set.

Returns Value associated with key.

Store/retrieve a setting in memory and persist during lifetime of application. To get the current value, only specify the key. To set a new value, call with key and new value.

clear_toggles()
Clears all state associated with the *toggle()* function.

clear_settings()
Clears all state associated with the *setting()* function.

MATH

randomrange(*start*[, *stop=None*[, *step=None*]])

Parameters

- **start** (*int*) – Start of range (inclusive)
- **end** (*int*) – End of range(not inclusive)
- **step** (*int*) – Interval at which to return a value.

Return a random integer between `[start, end)`.

gauss_distribution(*mean*, *sigma*)

Parameters

- **mean** (*float*) – Mean value
- **sigma** (*float*) – Standard deviation

sqrt(*n*)
Calculate the square root of `n`.

tonumber(*s*)

Parameters **s** (*str*) – String to convert to number.

Returns Integer, floating-point or NULL on failure.

mode(*val*)

Parameters **val** – Numbers in list.

Returns The mode, or most-common, number observed.

Aggregate function which calculates *mode* of values.

minrange(*val*)

Parameters **val** – Value

Returns Min difference between two values.

Aggregate function which calculates the minimal distance between two numbers in the sequence.

avgrange(*val*)

Parameters **val** – Value

Returns Average difference between values.

Aggregate function which calculates the average distance between two consecutive numbers in the sequence.

range(*val*)

Parameters **val** – Value

Returns The range from the smallest to largest value in sequence.

Aggregate function which returns range of values observed.

median(*val*)

Parameters **val** – Value

Returns The median, or middle, value in a sequence.

Aggregate function which calculates the middle value in a sequence.

Note: Only available if you compiled the `_sqlite_udf` extension.

STRING

substr_count(*haystack*, *needle*)
Returns number of times `needle` appears in `haystack`.

strip_chars(*haystack*, *chars*)
Strips any characters in `chars` from beginning and end of `haystack`.

damerau_levenshtein_dist(*s1*, *s2*)
Computes the edit distance from s1 to s2 using the damerau variant of the levenshtein algorithm.

Note: Only available if you compiled the `_sqlite_udf` extension.

---

levenshtein_dist(*s1*, *s2*)
Computes the edit distance from s1 to s2 using the levenshtein algorithm.

Note: Only available if you compiled the `_sqlite_udf` extension.

---

str_dist(*s1*, *s2*)
Computes the edit distance from s1 to s2 using the standard library SequenceMatcher's algorithm.

Note: Only available if you compiled the `_sqlite_udf` extension.

regex_search(*regex*, *search_string*)

Parameters

- **regex** (*str*) – Regular expression
- **search_string** (*str*) – String to search for instances of regex.

Table-value function that searches a string for substrings that match the provided `regex`. Returns rows for each match found.

Example:

```
SELECT * FROM regex_search('\w+', 'extract words, ignore! symbols');

value
-----
extract
words
ignore
symbols
```

1.13.4 apsw, an advanced sqlite driver

The `apsw_ext` module contains a database class suitable for use with the apsw sqlite driver.

APSW Project page: https://github.com/rogerbinns/apsw

APSW is a really neat library that provides a thin wrapper on top of SQLite's C interface, making it possible to use all of SQLite's advanced features.

Here are just a few reasons to use APSW, taken from the documentation:

- APSW gives all functionality of SQLite, including virtual tables, virtual file system, blob i/o, backups and file control.
- Connections can be shared across threads without any additional locking.
- Transactions are managed explicitly by your code.
- APSW can handle nested transactions.
- Unicode is handled correctly.
- APSW is faster.

For more information on the differences between apsw and pysqlite, check the apsw docs.

How to use the APSWDatabase

```
from apsw_ext import *

db = APSWDatabase(':memory:')

class BaseModel(Model):
    class Meta:
```

(continues on next page)

(continued from previous page)

```
        database = db

class SomeModel(BaseModel):
    col1 = CharField()
    col2 = DateTimeField()
```

apsw_ext API notes

APSWDatabase extends the *SqliteExtDatabase* and inherits its advanced features.

class APSWDatabase(*database*, ***connect_kwargs*)

Parameters

- **database** (*string*) – filename of sqlite database
- **connect_kwargs** – keyword arguments passed to apsw when opening a connection

register_module(*mod_name*, *mod_inst*)

Provides a way of globally registering a module. For more information, see the documentation on virtual tables.

Parameters

- **mod_name** (*string*) – name to use for module
- **mod_inst** (*object*) – an object implementing the Virtual Table interface

unregister_module(*mod_name*)

Unregister a module.

Parameters **mod_name** (*string*) – name to use for module

Note: Be sure to use the `Field` subclasses defined in the `apsw_ext` module, as they will properly handle adapting the data types for storage.

For example, instead of using `peewee.DateTimeField`, be sure you are importing and using `playhouse.apsw_ext.DateTimeField`.

1.13.5 Sqlcipher backend

- Although this extention's code is short, it has not been properly peer-reviewed yet and may have introduced vulnerabilities.

Also note that this code relies on pysqlcipher and sqlcipher, and the code there might have vulnerabilities as well, but since these are widely used crypto modules, we can expect "short zero days" there.

sqlcipher_ext API notes

class SqlCipherDatabase(*database*, *passphrase*, ***kwargs*)

Subclass of *SqliteDatabase* that stores the database encrypted. Instead of the standard `sqlite3` backend, it uses pysqlcipher: a python wrapper for sqlcipher, which – in turn – is an encrypted wrapper around `sqlite3`, so the API is *identical* to *SqliteDatabase*'s, except for object construction parameters:

Parameters

- **database** – Path to encrypted database filename to open [or create].
- **passphrase** – Database encryption passphrase: should be at least 8 character long, but it is *strongly advised* to enforce better passphrase strength criteria in your implementation.

- If the `database` file doesn't exist, it will be *created* with encryption by a key derived from `passhprase`.
- When trying to open an existing database, `passhprase` should be identical to the ones used when it was created. If the passphrase is incorrect, an error will be raised when first attempting to access the database.

rekey(*passphrase*)

Parameters **passphrase** (*str*) – New passphrase for database.

Change the passphrase for database.

Note: SQLCipher can be configured using a number of extension PRAGMAs. The list of PRAGMAs and their descriptions can be found in the SQLCipher documentation.

For example to specify the number of PBKDF2 iterations for the key derivation (64K in SQLCipher 3.x, 256K in SQLCipher 4.x by default):

```
# Use 1,000,000 iterations.
db = SqlCipherDatabase('my_app.db', pragmas={'kdf_iter': 1000000})
```

To use a cipher page-size of 16KB and a cache-size of 10,000 pages:

```
db = SqlCipherDatabase('my_app.db', passphrase='secret!!!', pragmas={
    'cipher_page_size': 1024 * 16,
    'cache_size': 10000})  # 10,000 16KB pages, or 160MB.
```

Example of prompting the user for a passphrase:

```
db = SqlCipherDatabase(None)

class BaseModel(Model):
    """Parent for all app's models"""
    class Meta:
        # We won't have a valid db until user enters passhrase.
        database = db

# Derive our model subclasses
class Person(BaseModel):
    name = TextField(primary_key=True)

right_passphrase = False
while not right_passphrase:
    db.init(
        'testsqlcipher.db',
        passphrase=get_passphrase_from_user())

    try:  # Actually execute a query against the db to test passphrase.
        db.get_tables()
    except DatabaseError as exc:
        # This error indicates the password was wrong.
        if exc.args[0] == 'file is encrypted or is not a database':
```

(continues on next page)

(continued from previous page)

```
            tell_user_the_passphrase_was_wrong()
            db.init(None)  # Reset the db.
        else:
            raise exc
    else:
        # The password was correct.
        right_passphrase = True
```

See also: a slightly more elaborate example.

1.13.6 Postgresql Extensions

The postgresql extensions module provides a number of "postgres-only" functions, currently:

- *json support*, including *jsonb* for Postgres 9.4.
- *hstore support*
- *server-side cursors*
- *full-text search*
- `ArrayField` field type, for storing arrays.
- `HStoreField` field type, for storing key/value pairs.
- `IntervalField` field type, for storing `timedelta` objects.
- `JSONField` field type, for storing JSON data.
- `BinaryJSONField` field type for the `jsonb` JSON data type.
- `TSVectorField` field type, for storing full-text search data.
- `DateTimeTZ` field type, a timezone-aware datetime field.

In the future I would like to add support for more of postgresql's features. If there is a particular feature you would like to see added, please open a Github issue.

Warning: In order to start using the features described below, you will need to use the extension `PostgresqlExtDatabase` class instead of `PostgresqlDatabase`.

The code below will assume you are using the following database and base model:

```
from playhouse.postgres_ext import *

ext_db = PostgresqlExtDatabase('peewee_test', user='postgres')

class BaseExtModel(Model):
    class Meta:
        database = ext_db
```

JSON Support

peewee has basic support for Postgres' native JSON data type, in the form of `JSONField`. As of version 2.4.7, peewee also supports the Postgres 9.4 binary json `jsonb` type, via `BinaryJSONField`.

Warning: Postgres supports a JSON data type natively as of 9.2 (full support in 9.3). In order to use this functionality you must be using the correct version of Postgres with *psycopg2* version 2.5 or greater.

To use *BinaryJSONField*, which has many performance and querying advantages, you must have Postgres 9.4 or later.

Note: You must be sure your database is an instance of *PostgresqlExtDatabase* in order to use the *JSONField*.

Here is an example of how you might declare a model with a JSON field:

```
import json
import urllib2
from playhouse.postgres_ext import *

db = PostgresqlExtDatabase('my_database')

class APIResponse(Model):
    url = CharField()
    response = JSONField()

    class Meta:
        database = db

    @classmethod
    def request(cls, url):
        fh = urllib2.urlopen(url)
        return cls.create(url=url, response=json.loads(fh.read()))

APIResponse.create_table()

# Store a JSON response.
offense = APIResponse.request('http://crime-api.com/api/offense/')
booking = APIResponse.request('http://crime-api.com/api/booking/')

# Query a JSON data structure using a nested key lookup:
offense_responses = APIResponse.select().where(
    APIResponse.response['meta']['model'] == 'offense')

# Retrieve a sub-key for each APIResponse. By calling .as_json(), the
# data at the sub-key will be returned as Python objects (dicts, lists,
# etc) instead of serialized JSON.
q = (APIResponse
     .select(
       APIResponse.data['booking']['person'].as_json().alias('person'))
     .where(APIResponse.data['meta']['model'] == 'booking'))

for result in q:
    print(result.person['name'], result.person['dob'])
```

The *BinaryJSONField* works the same and supports the same operations as the regular *JSONField*, but provides several additional operations for testing **containment**. Using the binary json field, you can test whether your JSON data contains other partial JSON structures (*contains()*, *contains_any()*, *contains_all()*), or whether it is a subset of a larger JSON document (*contained_by()*).

For more examples, see the *JSONField* and *BinaryJSONField* API documents below.

hstore support

Postgresql hstore is an embedded key/value store. With hstore, you can store arbitrary key/value pairs in your database alongside structured relational data.

To use `hstore`, you need to specify an additional parameter when instantiating your *`PostgresqlExtDatabase`*:

```
# Specify "register_hstore=True":
db = PostgresqlExtDatabase('my_db', register_hstore=True)
```

Currently the `postgres_ext` module supports the following operations:

- Store and retrieve arbitrary dictionaries
- Filter by key(s) or partial dictionary
- Update/add one or more keys to an existing dictionary
- Delete one or more keys from an existing dictionary
- Select keys, values, or zip keys and values
- Retrieve a slice of keys/values
- Test for the existence of a key
- Test that a key has a non-NULL value

Using hstore

To start with, you will need to import the custom database class and the hstore functions from `playhouse.postgres_ext` (see above code snippet). Then, it is as simple as adding a *`HStoreField`* to your model:

```
class House(BaseExtModel):
    address = CharField()
    features = HStoreField()
```

You can now store arbitrary key/value pairs on `House` instances:

```
>>> h = House.create(
...     address='123 Main St',
...     features={'garage': '2 cars', 'bath': '2 bath'})
...
>>> h_from_db = House.get(House.id == h.id)
>>> h_from_db.features
{'bath': '2 bath', 'garage': '2 cars'}
```

You can filter by individual key, multiple keys or partial dictionary:

```
>>> query = House.select()
>>> garage = query.where(House.features.contains('garage'))
>>> garage_and_bath = query.where(House.features.contains(['garage', 'bath']))
>>> twocar = query.where(House.features.contains({'garage': '2 cars'}))
```

Suppose you want to do an atomic update to the house:

```
>>> new_features = House.features.update({'bath': '2.5 bath', 'sqft': '1100'})
>>> query = House.update(features=new_features)
>>> query.where(House.id == h.id).execute()
```

(continues on next page)

(continued from previous page)

```
1
>>> h = House.get(House.id == h.id)
>>> h.features
{'bath': '2.5 bath', 'garage': '2 cars', 'sqft': '1100'}
```

Or, alternatively an atomic delete:

```
>>> query = House.update(features=House.features.delete('bath'))
>>> query.where(House.id == h.id).execute()
1
>>> h = House.get(House.id == h.id)
>>> h.features
{'garage': '2 cars', 'sqft': '1100'}
```

Multiple keys can be deleted at the same time:

```
>>> query = House.update(features=House.features.delete('garage', 'sqft'))
```

You can select just keys, just values, or zip the two:

```
>>> for h in House.select(House.address, House.features.keys().alias('keys')):
...     print(h.address, h.keys)

123 Main St [u'bath', u'garage']

>>> for h in House.select(House.address, House.features.values().alias('vals')):
...     print(h.address, h.vals)

123 Main St [u'2 bath', u'2 cars']

>>> for h in House.select(House.address, House.features.items().alias('mtx')):
...     print(h.address, h.mtx)

123 Main St [[u'bath', u'2 bath'], [u'garage', u'2 cars']]
```

You can retrieve a slice of data, for example, all the garage data:

```
>>> query = House.select(House.address, House.features.slice('garage').alias('garage_
↪data'))
>>> for house in query:
...     print(house.address, house.garage_data)

123 Main St {'garage': '2 cars'}
```

You can check for the existence of a key and filter rows accordingly:

```
>>> has_garage = House.features.exists('garage')
>>> for house in House.select(House.address, has_garage.alias('has_garage')):
...     print(house.address, house.has_garage)

123 Main St True

>>> for house in House.select().where(House.features.exists('garage')):
...     print(house.address, house.features['garage'])  # <-- just houses w/garage␣
↪data

123 Main St 2 cars
```

Interval support

Postgres supports durations through the `INTERVAL` data-type (docs).

class IntervalField([*null=False*[, ...]])

Field class capable of storing Python `datetime.timedelta` instances.

Example:

```
from datetime import timedelta

from playhouse.postgres_ext import *

db = PostgresqlExtDatabase('my_db')

class Event(Model):
    location = CharField()
    duration = IntervalField()
    start_time = DateTimeField()

    class Meta:
        database = db

    @classmethod
    def get_long_meetings(cls):
        return cls.select().where(cls.duration > timedelta(hours=1))
```

Server-side cursors

When psycopg2 executes a query, normally all results are fetched and returned to the client by the backend. This can cause your application to use a lot of memory when making large queries. Using server-side cursors, results are returned a little at a time (by default 2000 records). For the definitive reference, please see the psycopg2 documentation.

Note: To use server-side (or named) cursors, you must be using *PostgresqlExtDatabase*.

To execute a query using a server-side cursor, simply wrap your select query using the *ServerSide()* helper:

```
large_query = PageView.select()  # Build query normally.

# Iterate over large query inside a transaction.
for page_view in ServerSide(large_query):
    # do some interesting analysis here.
    pass

# Server-side resources are released.
```

If you would like all `SELECT` queries to automatically use a server-side cursor, you can specify this when creating your *PostgresqlExtDatabase*:

```
from postgres_ext import PostgresqlExtDatabase

ss_db = PostgresqlExtDatabase('my_db', server_side_cursors=True)
```

Note: Server-side cursors live only as long as the transaction, so for this reason peewee will not automatically call `commit()` after executing a `SELECT` query. If you do not `commit` after you are done iterating, you will not release

the server-side resources until the connection is closed (or the transaction is committed later). Furthermore, since peewee will by default cache rows returned by the cursor, you should always call `.iterator()` when iterating over a large query.

If you are using the *ServerSide()* helper, the transaction and call to `iterator()` will be handled transparently.

Full-text search

Postgresql provides sophisticated full-text search using special data-types (`tsvector` and `tsquery`). Documents should be stored or converted to the `tsvector` type, and search queries should be converted to `tsquery`.

For simple cases, you can simply use the *Match()* function, which will automatically perform the appropriate conversions, and requires no schema changes:

```
def blog_search(search_term):
    return Blog.select().where(
        (Blog.status == Blog.STATUS_PUBLISHED) &
        Match(Blog.content, search_term))
```

The *Match()* function will automatically convert the left-hand operand to a `tsvector`, and the right-hand operand to a `tsquery`. For better performance, it is recommended you create a `GIN` index on the column you plan to search:

```
CREATE INDEX blog_full_text_search ON blog USING gin(to_tsvector(content));
```

Alternatively, you can use the *TSVectorField* to maintain a dedicated column for storing `tsvector` data:

```
class Blog(Model):
    content = TextField()
    search_content = TSVectorField()
```

Note: *TSVectorField*, will automatically be created with a GIN index.

You will need to explicitly convert the incoming text data to `tsvector` when inserting or updating the `search_content` field:

```
content = 'Excellent blog post about peewee ORM.'
blog_entry = Blog.create(
    content=content,
    search_content=fn.to_tsvector(content))
```

To perform a full-text search, use *TSVectorField.match()*:

```
terms = 'python & (sqlite | postgres)'
results = Blog.select().where(Blog.search_content.match(terms))
```

For more information, see the Postgres full-text search docs.

postgres_ext API notes

class PostgresqlExtDatabase(*database*[, *server_side_cursors=False*[, *register_hstore=False*[, ...]]])

Identical to *PostgresqlDatabase* but required in order to support:

Parameters

- **database** (*str*) – Name of database to connect to.
- **server_side_cursors** (*bool*) – Whether `SELECT` queries should utilize server-side cursors.
- **register_hstore** (*bool*) – Register the HStore extension with the connection.

- *Server-side cursors*
- *ArrayField*
- *DateTimeTZField*
- *JSONField*
- *BinaryJSONField*
- *HStoreField*
- *TSVectorField*

If you wish to use the HStore extension, you must specify `register_hstore=True`.

If using `server_side_cursors`, also be sure to wrap your queries with *ServerSide()*.

ServerSide(*select_query*)

Parameters **select_query** – a *SelectQuery* instance.

Rtype generator

Wrap the given select query in a transaction, and call its `iterator()` method to avoid caching row instances. In order for the server-side resources to be released, be sure to exhaust the generator (iterate over all the rows).

Usage:

```
large_query = PageView.select()
for page_view in ServerSide(large_query):
    # Do something interesting.
    pass

# At this point server side resources are released.
```

class ArrayField([*field_class=IntegerField*[, *field_kwargs=None*[, *dimensions=1*[, *convert_values=False*]]]])

Parameters

- **field_class** – a subclass of *Field*, e.g. *IntegerField*.
- **field_kwargs** (*dict*) – arguments to initialize `field_class`.
- **dimensions** (*int*) – dimensions of array.
- **convert_values** (*bool*) – apply `field_class` value conversion to array data.

Field capable of storing arrays of the provided *field_class*.

Note: By default ArrayField will use a GIN index. To disable this, initialize the field with `index=False`.

You can store and retrieve lists (or lists-of-lists):

```
class BlogPost(BaseModel):
    content = TextField()
    tags = ArrayField(CharField)


post = BlogPost(content='awesome', tags=['foo', 'bar', 'baz'])
```

Additionally, you can use the __getitem__ API to query values or slices in the database:

```
# Get the first tag on a given blog post.
first_tag = (BlogPost
             .select(BlogPost.tags[0].alias('first_tag'))
             .where(BlogPost.id == 1)
             .dicts()
             .get())

# first_tag = {'first_tag': 'foo'}
```

Get a slice of values:

```
# Get the first two tags.
two_tags = (BlogPost
            .select(BlogPost.tags[:2].alias('two'))
            .dicts()
            .get())
# two_tags = {'two': ['foo', 'bar']}
```

contains(**items*)

Parameters **items** – One or more items that must be in the given array field.

```
# Get all blog posts that are tagged with both "python" and "django".
Blog.select().where(Blog.tags.contains('python', 'django'))
```

contains_any(**items*)

Parameters **items** – One or more items to search for in the given array field.

Like *contains()*, except will match rows where the array contains *any* of the given items.

```
# Get all blog posts that are tagged with "flask" and/or "django".
Blog.select().where(Blog.tags.contains_any('flask', 'django'))
```

class DateTimeTZField(**args*, ***kwargs*)

A timezone-aware subclass of *DateTimeField*.

class HStoreField(**args*, ***kwargs*)

A field for storing and retrieving arbitrary key/value pairs. For details on usage, see *hstore support*.

Attention: To use the *HStoreField* you will need to be sure the *hstore* extension is registered with the connection. To accomplish this, instantiate the *PostgresqlExtDatabase* with `register_hstore=True`.

Note: By default `HStoreField` will use a *GiST* index. To disable this, initialize the field with `index=False`.

keys()
Returns the keys for a given row.

```
>>> for h in House.select(House.address, House.features.keys().alias('keys')):
...     print(h.address, h.keys)

123 Main St [u'bath', u'garage']
```

values()
Return the values for a given row.

```
>>> for h in House.select(House.address, House.features.values().alias('vals
↪')):
...     print(h.address, h.vals)

123 Main St [u'2 bath', u'2 cars']
```

items()
Like python's `dict`, return the keys and values in a list-of-lists:

```
>>> for h in House.select(House.address, House.features.items().alias('mtx')):
...     print(h.address, h.mtx)

123 Main St [[u'bath', u'2 bath'], [u'garage', u'2 cars']]
```

slice(**args*)
Return a slice of data given a list of keys.

```
>>> for h in House.select(House.address, House.features.slice('garage').alias(
↪'garage_data')):
...     print(h.address, h.garage_data)

123 Main St {'garage': '2 cars'}
```

exists(*key*)
Query for whether the given key exists.

```
>>> for h in House.select(House.address, House.features.exists('garage').
↪alias('has_garage')):
...     print(h.address, h.has_garage)

123 Main St True

>>> for h in House.select().where(House.features.exists('garage')):
...     print(h.address, h.features['garage']) # <-- just houses w/garage data

123 Main St 2 cars
```

defined(*key*)
Query for whether the given key has a value associated with it.

update(***data*)
Perform an atomic update to the keys/values for a given row or rows.

```
>>> query = House.update(features=House.features.update(
...     sqft=2000,
...     year_built=2012))
>>> query.where(House.id == 1).execute()
```

delete(**keys*)

Delete the provided keys for a given row or rows.

> **Note:** We will use an `UPDATE` query.

```
>>> query = House.update(features=House.features.delete(
...     'sqft', 'year_built'))
>>> query.where(House.id == 1).execute()
```

contains(*value*)

Parameters **value** – Either a `dict`, a `list` of keys, or a single key.

Query rows for the existence of either:

- a partial dictionary.
- a list of keys.
- a single key.

```
>>> query = House.select()
>>> has_garage = query.where(House.features.contains('garage'))
>>> garage_bath = query.where(House.features.contains(['garage', 'bath']))
>>> twocar = query.where(House.features.contains({'garage': '2 cars'}))
```

contains_any(**keys*)

Parameters **keys** – One or more keys to search for.

Query rows for the existence of *any* key.

class JSONField(*dumps=None*, **args*, ***kwargs*)

Parameters **dumps** – The default is to call json.dumps() or the dumps function. You can override this method to create a customized JSON wrapper.

Field class suitable for storing and querying arbitrary JSON. When using this on a model, set the field's value to a Python object (either a `dict` or a `list`). When you retrieve your value from the database it will be returned as a Python data structure.

> **Note:** You must be using Postgres 9.2 / psycopg2 2.5 or greater.

> **Note:** If you are using Postgres 9.4, strongly consider using the *BinaryJSONField* instead as it offers better performance and more powerful querying options.

Example model declaration:

```
db = PostgresqlExtDatabase('my_db')

class APIResponse(Model):
    url = CharField()
    response = JSONField()
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        database = db
```

Example of storing JSON data:

```
url = 'http://foo.com/api/resource/'
resp = json.loads(urllib2.urlopen(url).read())
APIResponse.create(url=url, response=resp)

APIResponse.create(url='http://foo.com/baz/', response={'key': 'value'})
```

To query, use Python's `[]` operators to specify nested key or array lookups:

```
APIResponse.select().where(
    APIResponse.response['key1']['nested-key'] == 'some-value')
```

To illustrate the use of the `[]` operators, imagine we have the following data stored in an `APIResponse`:

```
{
  "foo": {
    "bar": ["i1", "i2", "i3"],
    "baz": {
      "huey": "mickey",
      "peewee": "nugget"
    }
  }
}
```

Here are the results of a few queries:

```
def get_data(expression):
    # Helper function to just retrieve the results of a
    # particular expression.
    query = (APIResponse
             .select(expression.alias('my_data'))
             .dicts()
             .get())
    return query['my_data']

# Accessing the foo -> bar subkey will return a JSON
# representation of the list.
get_data(APIResponse.data['foo']['bar'])
# '["i1", "i2", "i3"]'

# In order to retrieve this list as a Python list,
# we will call .as_json() on the expression.
get_data(APIResponse.data['foo']['bar'].as_json())
# ['i1', 'i2', 'i3']

# Similarly, accessing the foo -> baz subkey will
# return a JSON representation of the dictionary.
get_data(APIResponse.data['foo']['baz'])
# '{"huey": "mickey", "peewee": "nugget"}'

# Again, calling .as_json() will return an actual
# python dictionary.
```

(continues on next page)

(continued from previous page)

```
get_data(APIResponse.data['foo']['baz'].as_json())
# {'huey': 'mickey', 'peewee': 'nugget'}

# When dealing with simple values, either way works as
# you expect.
get_data(APIResponse.data['foo']['bar'][0])
# 'i1'

# Calling .as_json() when the result is a simple value
# will return the same thing as the previous example.
get_data(APIResponse.data['foo']['bar'][0].as_json())
# 'i1'
```

class BinaryJSONField(*dumps=None*, **args*, ***kwargs*)

Parameters **dumps** – The default is to call json.dumps() or the dumps function. You can override this method to create a customized JSON wrapper.

Store and query arbitrary JSON documents. Data should be stored using normal Python `dict` and `list` objects, and when data is returned from the database, it will be returned using `dict` and `list` as well.

For examples of basic query operations, see the above code samples for *JSONField*. The example queries below will use the same `APIResponse` model described above.

Note: By default BinaryJSONField will use a GiST index. To disable this, initialize the field with `index=False`.

Note: You must be using Postgres 9.4 / psycopg2 2.5 or newer. If you are using Postgres 9.2 or 9.3, you can use the regular *JSONField* instead.

---

contains(*other*)

Test whether the given JSON data contains the given JSON fragment or key.

Example:

```
search_fragment = {
    'foo': {'bar': ['i2']}
}
query = (APIResponse
         .select()
         .where(APIResponse.data.contains(search_fragment)))

# If we're searching for a list, the list items do not need to
# be ordered in a particular way:
query = (APIResponse
         .select()
         .where(APIResponse.data.contains({
             'foo': {'bar': ['i2', 'i1']}})))
```

We can pass in simple keys as well. To find APIResponses that contain the key `foo` at the top-level:

```
APIResponse.select().where(APIResponse.data.contains('foo'))
```

We can also search sub-keys using square-brackets:

```
APIResponse.select().where(
    APIResponse.data['foo']['bar'].contains(['i2', 'i1']))
```

contains_any(**items*)

Search for the presence of one or more of the given items.

```
APIResponse.select().where(
    APIResponse.data.contains_any('foo', 'baz', 'nugget'))
```

Like *contains()*, we can also search sub-keys:

```
APIResponse.select().where(
    APIResponse.data['foo']['bar'].contains_any('i2', 'ix'))
```

contains_all(**items*)

Search for the presence of all of the given items.

```
APIResponse.select().where(
    APIResponse.data.contains_all('foo'))
```

Like *contains_any()*, we can also search sub-keys:

```
APIResponse.select().where(
    APIResponse.data['foo']['bar'].contains_all('i1', 'i2', 'i3'))
```

contained_by(*other*)

Test whether the given JSON document is contained by (is a subset of) the given JSON document. This method is the inverse of *contains()*.

```
big_doc = {
    'foo': {
        'bar': ['i1', 'i2', 'i3'],
        'baz': {
            'huey': 'mickey',
            'peewee': 'nugget',
        }
    },
    'other_key': ['nugget', 'bear', 'kitten'],
}
APIResponse.select().where(
    APIResponse.data.contained_by(big_doc))
```

concat(*data*)

Concatentate two field data and the provided data. Note that this operation does not merge or do a "deep concat".

has_key(*key*)

Test whether the key exists at the top-level of the JSON object.

remove(**keys*)

Remove one or more keys from the top-level of the JSON object.

Match(*field*, *query*)

Generate a full-text search expression, automatically converting the left-hand operand to a `tsvector`, and the right-hand operand to a `tsquery`.

Example:

```
def blog_search(search_term):
    return Blog.select().where(
        (Blog.status == Blog.STATUS_PUBLISHED) &
        Match(Blog.content, search_term))
```

class TSVectorField

Field type suitable for storing `tsvector` data. This field will automatically be created with a `GIN` index for improved search performance.

Note: Data stored in this field will still need to be manually converted to the `tsvector` type.

Note:

By default TSVectorField will use a GIN index. To disable this, initialize the field with `index=False`.

Example usage:

```
class Blog(Model):
    content = TextField()
    search_content = TSVectorField()

content = 'this is a sample blog entry.'
blog_entry = Blog.create(
    content=content,
    search_content=fn.to_tsvector(content))  # Note `to_tsvector()`.
```

---

match(*query*[, *language=None*[, *plain=False*]])

Parameters

- **query** (*str*) – the full-text search query.
- **language** (*str*) – language name (optional).
- **plain** (*bool*) – parse search query using plain (simple) parser.

Returns an expression representing full-text search/match.

Example:

```
# Perform a search using the "match" method.
terms = 'python & (sqlite | postgres)'
results = Blog.select().where(Blog.search_content.match(terms))
```

1.13.7 Cockroach Database

CockroachDB (CRDB) is well supported by peewee.

```
from playhouse.cockroachdb import CockroachDatabase

db = CockroachDatabase('my_app', user='root', host='10.1.0.8')
```

The `playhouse.cockroachdb` extension module provides the following classes and helpers:

- *CockroachDatabase* - a subclass of *PostgresqlDatabase*, designed specifically for working with CRDB.
- *PooledCockroachDatabase* - like the above, but implements connection-pooling.
- *run_transaction()* - runs a function inside a transaction and provides automatic client-side retry logic.

Special field-types that may be useful when using CRDB:

- *UUIDKeyField* - a primary-key field implementation that uses CRDB's `UUID` type with a default randomly-generated UUID.
- *RowIDField* - a primary-key field implementation that uses CRDB's `INT` type with a default `unique_rowid()`.
- *JSONField* - same as the Postgres *BinaryJSONField*, as CRDB treats JSON as JSONB.
- *ArrayField* - same as the Postgres extension (but does not support multi-dimensional arrays).

CRDB is compatible with Postgres' wire protocol and exposes a very similar SQL interface, so it is possible (though **not recommended**) to use *PostgresqlDatabase* with CRDB:

1. CRDB does not support nested transactions (savepoints), so the *atomic()* method has been implemented to enforce this when using *CockroachDatabase*. For more info *CRDB Transactions*.
2. CRDB may have subtle differences in field-types, date functions and introspection from Postgres.
3. CRDB-specific features are exposed by the *CockroachDatabase*, such as specifying a transaction priority or the `AS OF SYSTEM TIME` clause.

CRDB Transactions

CRDB does not support nested transactions (savepoints), so the *atomic()* method on the *CockroachDatabase* has been modified to raise an exception if an invalid nesting is encountered. If you would like to be able to nest transactional code, you can use the *transaction()* method, which will ensure that the outer-most block will manage the transaction (e.g., exiting a nested-block will not cause an early commit).

Example:

```
@db.transaction()
def create_user(username):
    return User.create(username=username)

def some_other_function():
    with db.transaction() as txn:
        # do some stuff...

        # This function is wrapped in a transaction, but the nested
        # transaction will be ignored and folded into the outer
        # transaction, as we are already in a wrapped-block (via the
        # context manager).
        create_user('some_user@example.com')

        # do other stuff.

    # At this point we have exited the outer-most block and the transaction
    # will be committed.
    return
```

CRDB provides client-side transaction retries, which are available using a special *run_transaction()* helper. This helper method accepts a callable, which is responsible for executing any transactional statements that may need to be retried.

Simplest possible example of *run_transaction()*:

```
def create_user(email):
    # Callable that accepts a single argument (the database instance) and
    # which is responsible for executing the transactional SQL.
    def callback(db_ref):
        return User.create(email=email)

    return db.run_transaction(callback, max_attempts=10)

huey = create_user('huey@example.com')
```

Note: The `cockroachdb.ExceededMaxAttempts` exception will be raised if the transaction cannot be committed after the given number of attempts. If the SQL is mal-formed, violates a constraint, etc., then the function will raise the exception to the caller.

Example of using *run_transaction()* to implement client-side retries for a transaction that transfers an amount from one account to another:

```
from playhouse.cockroachdb import CockroachDatabase

db = CockroachDatabase('my_app')


def transfer_funds(from_id, to_id, amt):
    """
    Returns a 3-tuple of (success?, from balance, to balance). If there are
    not sufficient funds, then the original balances are returned.
    """
    def thunk(db_ref):
        src, dest = (Account
                     .select()
                     .where(Account.id.in_([from_id, to_id])))
        if src.id != from_id:
            src, dest = dest, src  # Swap order.

        # Cannot perform transfer, insufficient funds!
        if src.balance < amt:
            return False, src.balance, dest.balance

        # Update each account, returning the new balance.
        src, = (Account
                .update(balance=Account.balance - amt)
                .where(Account.id == from_id)
                .returning(Account.balance)
                .execute())
        dest, = (Account
                 .update(balance=Account.balance + amt)
                 .where(Account.id == to_id)
                 .returning(Account.balance)
                 .execute())
        return True, src.balance, dest.balance
```

(continues on next page)

(continued from previous page)

```
    # Perform the queries that comprise a logical transaction. In the
    # event the transaction fails due to contention, it will be auto-
    # matically retried (up to 10 times).
    return db.run_transaction(thunk, max_attempts=10)
```

CRDB APIs

class CockroachDatabase(*database*[, ***kwargs*])

CockroachDB implementation, based on the *PostgresqlDatabase* and using the `psycopg2` driver.

Additional keyword arguments are passed to the psycopg2 connection constructor, and may be used to specify the database `user`, `port`, etc.

run_transaction(*callback*[, *max_attempts=None*[, *system_time=None*[, *priority=None*]]])

Parameters

- **callback** – callable that accepts a single `db` parameter (which will be the database instance this method is called from).
- **max_attempts** (*int*) – max number of times to try before giving up.
- **system_time** (*datetime*) – execute the transaction `AS OF SYSTEM TIME` with respect to the given value.
- **priority** (*str*) – either "low", "normal" or "high".

Returns returns the value returned by the callback.

Raises `ExceededMaxAttempts` if `max_attempts` is exceeded.

Run SQL in a transaction with automatic client-side retries.

User-provided `callback`:

- **Must** accept one parameter, the `db` instance representing the connection the transaction is running under.
- **Must** not attempt to commit, rollback or otherwise manage the transaction.
- **May** be called more than one time.
- **Should** ideally only contain SQL operations.

Additionally, the database must not have any open transactions at the time this function is called, as CRDB does not support nested transactions. Attempting to do so will raise a `NotImplementedError`.

Simplest possible example:

```
def create_user(email):
    def callback(db_ref):
        return User.create(email=email)

    return db.run_transaction(callback, max_attempts=10)

user = create_user('huey@example.com')
```

class PooledCockroachDatabase(*database*[, ***kwargs*])

CockroachDB connection-pooling implementation, based on *PooledPostgresqlDatabase*. Implements the same APIs as *CockroachDatabase*, but will do client-side connection pooling.

run_transaction(*db*, *callback*[, *max_attempts=None*[, *system_time=None*[, *priority=None*]]])
Run SQL in a transaction with automatic client-side retries. See *CockroachDatabase.run_transaction()* for details.

Parameters

- **db** (CockroachDatabase) – database instance.
- **callback** – callable that accepts a single db parameter (which will be the same as the value passed above).

Note: This function is equivalent to the identically-named method on the *CockroachDatabase* class.

class UUIDKeyField
UUID primary-key field that uses the CRDB gen_random_uuid() function to automatically populate the initial value.

class RowIDField
Auto-incrementing integer primary-key field that uses the CRDB unique_rowid() function to automatically populate the initial value.

See also:

- *BinaryJSONField* from the Postgresql extension (available in the cockroachdb extension module, and aliased to JSONField).
- *ArrayField* from the Postgresql extension.

1.13.8 MySQL Extensions

Peewee provides an alternate database implementation for using the mysql-connector driver. The implementation can be found in playhouse.mysql_ext.

Example usage:

```
from playhouse.mysql_ext import MySQLConnectorDatabase

# MySQL database implementation that utilizes mysql-connector driver.
db = MySQLConnectorDatabase('my_database', host='1.2.3.4', user='mysql')
```

Additional MySQL-specific helpers:

class JSONField
Extends *TextField* and implements transparent JSON encoding and decoding in Python.

Match(*columns*, *expr*[, *modifier=None*])

Parameters

- **columns** – a single *Field* or a tuple of multiple fields.
- **expr** (*str*) – the full-text search expression.
- **modifier** (*str*) – optional modifiers for the search, e.g. *'in boolean mode'*.

Helper class for constructing MySQL full-text search queries of the form:

```
MATCH (columns, ...) AGAINST (expr[ modifier])
```

1.13.9 DataSet

The *dataset* module contains a high-level API for working with databases modeled after the popular project of the same name. The aims of the *dataset* module are to provide:

- A simplified API for working with relational data, along the lines of working with JSON.
- An easy way to export relational data as JSON or CSV.
- An easy way to import JSON or CSV data into a relational database.

A minimal data-loading script might look like this:

```
from playhouse.dataset import DataSet

db = DataSet('sqlite:///:memory:')

table = db['sometable']
table.insert(name='Huey', age=3)
table.insert(name='Mickey', age=5, gender='male')

huey = table.find_one(name='Huey')
print(huey)
# {'age': 3, 'gender': None, 'id': 1, 'name': 'Huey'}

for obj in table:
    print(obj)
# {'age': 3, 'gender': None, 'id': 1, 'name': 'Huey'}
# {'age': 5, 'gender': 'male', 'id': 2, 'name': 'Mickey'}
```

You can insert, update or delete using the dictionary APIs as well:

```
huey = table.find_one(name='Huey')
# {'age': 3, 'gender': None, 'id': 1, 'name': 'Huey'}

# Perform an update by supplying a partial record of changes.
table[1] = {'gender': 'male', 'age': 4}
print(table[1])
# {'age': 4, 'gender': 'male', 'id': 1, 'name': 'Huey'}

# Or insert a new record:
table[3] = {'name': 'Zaizee', 'age': 2}
print(table[3])
# {'age': 2, 'gender': None, 'id': 3, 'name': 'Zaizee'}

# Or delete a record:
del table[3]  # Remove the row we just added.
```

You can export or import data using *freeze()* and *thaw()*:

```
# Export table content to the `users.json` file.
db.freeze(table.all(), format='json', filename='users.json')

# Import data from a CSV file into a new table. Columns will be automatically
# created for each field in the CSV file.
new_table = db['stats']
new_table.thaw(format='csv', filename='monthly_stats.csv')
```

Getting started

DataSet objects are initialized by passing in a database URL of the format `dialect://user:password@host/dbname`. See the *Database URL* section for examples of connecting to various databases.

```
# Create an in-memory SQLite database.
db = DataSet('sqlite:///:memory:')
```

Storing data

To store data, we must first obtain a reference to a table. If the table does not exist, it will be created automatically:

```
# Get a table reference, creating the table if it does not exist.
table = db['users']
```

We can now *insert()* new rows into the table. If the columns do not exist, they will be created automatically:

```
table.insert(name='Huey', age=3, color='white')
table.insert(name='Mickey', age=5, gender='male')
```

To update existing entries in the table, pass in a dictionary containing the new values and filter conditions. The list of columns to use as filters is specified in the *columns* argument. If no filter columns are specified, then all rows will be updated.

```
# Update the gender for "Huey".
table.update(name='Huey', gender='male', columns=['name'])

# Update all records. If the column does not exist, it will be created.
table.update(favorite_orm='peewee')
```

Importing data

To import data from an external source, such as a JSON or CSV file, you can use the *thaw()* method. By default, new columns will be created for any attributes encountered. If you wish to only populate columns that are already defined on a table, you can pass in `strict=True`.

```
# Load data from a JSON file containing a list of objects.
table = dataset['stock_prices']
table.thaw(filename='stocks.json', format='json')
table.all()[:3]

# Might print...
[{'id': 1, 'ticker': 'GOOG', 'price': 703},
 {'id': 2, 'ticker': 'AAPL', 'price': 109},
 {'id': 3, 'ticker': 'AMZN', 'price': 300}]
```

Using transactions

DataSet supports nesting transactions using a simple context manager.

```
table = db['users']
with db.transaction() as txn:
    table.insert(name='Charlie')

    with db.transaction() as nested_txn:
        # Set Charlie's favorite ORM to Django.
        table.update(name='Charlie', favorite_orm='django', columns=['name'])

        # jk/lol
        nested_txn.rollback()
```

Inspecting the database

You can use the `tables()` method to list the tables in the current database:

```
>>> print db.tables
['sometable', 'user']
```

And for a given table, you can print the columns:

```
>>> table = db['user']
>>> print table.columns
['id', 'age', 'name', 'gender', 'favorite_orm']
```

We can also find out how many rows are in a table:

```
>>> print len(db['user'])
3
```

Reading data

To retrieve all rows, you can use the *all()* method:

```
# Retrieve all the users.
users = db['user'].all()

# We can iterate over all rows without calling `.all()`
for user in db['user']:
    print user['name']
```

Specific objects can be retrieved using *find()* and *find_one()*.

```
# Find all the users who like peewee.
peewee_users = db['user'].find(favorite_orm='peewee')

# Find Huey.
huey = db['user'].find_one(name='Huey')
```

Exporting data

To export data, use the *freeze()* method, passing in the query you wish to export:

```
peewee_users = db['user'].find(favorite_orm='peewee')
db.freeze(peewee_users, format='json', filename='peewee_users.json')
```

API

class DataSet(*url*)

Parameters **url** – A database URL or a *Database* instance. For details on using a URL, see *Database URL* for examples.

The *DataSet* class provides a high-level API for working with relational databases.

tables
Return a list of tables stored in the database. This list is computed dynamically each time it is accessed.

__getitem__(*table_name*)
Provide a *Table* reference to the specified table. If the table does not exist, it will be created.

query(*sql*[, *params=None*[, *commit=True*]])

Parameters

- **sql** (*str*) – A SQL query.
- **params** (*list*) – Optional parameters for the query.
- **commit** (*bool*) – Whether the query should be committed upon execution.

Returns A database cursor.

Execute the provided query against the database.

transaction()
Create a context manager representing a new transaction (or savepoint).

freeze(*query*[, *format='csv'*[, *filename=None*[, *file_obj=None*[, ***kwargs*]]]])

Parameters

- **query** – A *SelectQuery*, generated using *all()* or *~Table.find*.
- **format** – Output format. By default, *csv* and *json* are supported.
- **filename** – Filename to write output to.
- **file_obj** – File-like object to write output to.
- **kwargs** – Arbitrary parameters for export-specific functionality.

thaw(*table*[, *format='csv'*[, *filename=None*[, *file_obj=None*[, *strict=False*[, ***kwargs*]]]]])

Parameters

- **table** (*str*) – The name of the table to load data into.
- **format** – Input format. By default, *csv* and *json* are supported.
- **filename** – Filename to read data from.
- **file_obj** – File-like object to read data from.
- **strict** (*bool*) – Whether to store values for columns that do not already exist on the table.
- **kwargs** – Arbitrary parameters for import-specific functionality.

connect()
Open a connection to the underlying database. If a connection is not opened explicitly, one will be opened the first time a query is executed.

close()
Close the connection to the underlying database.

class Table(*dataset*, *name*, *model_class*)

Noindex

Provides a high-level API for working with rows in a given table.

columns
Return a list of columns in the given table.

model_class
A dynamically-created *Model* class.

create_index(*columns*[, *unique=False*])
Create an index on the given columns:

```
# Create a unique index on the `username` column.
db['users'].create_index(['username'], unique=True)
```

insert(***data*)
Insert the given data dictionary into the table, creating new columns as needed.

update(*columns=None*, *conjunction=None*, ***data*)
Update the table using the provided data. If one or more columns are specified in the *columns* parameter, then those columns' values in the *data* dictionary will be used to determine which rows to update.

```
# Update all rows.
db['users'].update(favorite_orm='peewee')

# Only update Huey's record, setting his age to 3.
db['users'].update(name='Huey', age=3, columns=['name'])
```

find(***query*)
Query the table for rows matching the specified equality conditions. If no query is specified, then all rows are returned.

```
peewee_users = db['users'].find(favorite_orm='peewee')
```

find_one(***query*)
Return a single row matching the specified equality conditions. If no matching row is found then `None` will be returned.

```
huey = db['users'].find_one(name='Huey')
```

all()
Return all rows in the given table.

delete(***query*)
Delete all rows matching the given equality conditions. If no query is provided, then all rows will be deleted.

```
# Adios, Django!
db['users'].delete(favorite_orm='Django')
```

(continues on next page)

(continued from previous page)

```
# Delete all the secret messages.
db['secret_messages'].delete()
```

freeze(*[format='csv'[, filename=None[, file_obj=None[, **kwargs]]]]*)

Parameters

- **format** – Output format. By default, *csv* and *json* are supported.
- **filename** – Filename to write output to.
- **file_obj** – File-like object to write output to.
- **kwargs** – Arbitrary parameters for export-specific functionality.

thaw(*[format='csv'[, filename=None[, file_obj=None[, strict=False[, **kwargs]]]]]*)

Parameters

- **format** – Input format. By default, *csv* and *json* are supported.
- **filename** – Filename to read data from.
- **file_obj** – File-like object to read data from.
- **strict** (*bool*) – Whether to store values for columns that do not already exist on the table.
- **kwargs** – Arbitrary parameters for import-specific functionality.

1.13.10 Fields

These fields can be found in the `playhouse.fields` module.

class CompressedField(*[compression_level=6[, algorithm='zlib'[, **kwargs]]]*)

Parameters

- **compression_level** (*int*) – A value from 0 to 9.
- **algorithm** (*str*) – Either `'zlib'` or `'bz2'`.

Stores compressed data using the specified algorithm. This field extends *BlobField*, transparently storing a compressed representation of the data in the database.

class PickleField

Stores arbitrary Python data by transparently pickling and un-pickling data stored in the field. This field extends *BlobField*. If the `cPickle` module is available, it will be used.

1.13.11 Hybrid Attributes

Hybrid attributes encapsulate functionality that operates at both the Python *and* SQL levels. The idea for hybrid attributes comes from a feature of the same name in SQLAlchemy. Consider the following example:

```
class Interval(Model):
    start = IntegerField()
    end = IntegerField()

    @hybrid_property
    def length(self):
```

(continues on next page)

(continued from previous page)

```
        return self.end - self.start

    @hybrid_method
    def contains(self, point):
        return (self.start <= point) & (point < self.end)
```

The *hybrid attribute* gets its name from the fact that the `length` attribute will behave differently depending on whether it is accessed via the `Interval` class or an `Interval` instance.

If accessed via an instance, then it behaves just as you would expect.

If accessed via the `Interval.length` class attribute, however, the length calculation will be expressed as a SQL expression. For example:

```
query = Interval.select().where(Interval.length > 5)
```

This query will be equivalent to the following SQL:

```
SELECT "t1"."id", "t1"."start", "t1"."end"
FROM "interval" AS t1
WHERE (("t1"."end" - "t1"."start") > 5)
```

The `playhouse.hybrid` module also contains a decorator for implementing hybrid methods which can accept parameters. As with hybrid properties, when accessed via a model instance, then the function executes normally as-written. When the hybrid method is called on the class, however, it will generate a SQL expression.

Example:

```
query = Interval.select().where(Interval.contains(2))
```

This query is equivalent to the following SQL:

```
SELECT "t1"."id", "t1"."start", "t1"."end"
FROM "interval" AS t1
WHERE (("t1"."start" <= 2) AND (2 < "t1"."end"))
```

There is an additional API for situations where the python implementation differs slightly from the SQL implementation. Let's add a `radius` method to the `Interval` model. Because this method calculates an absolute value, we will use the Python `abs()` function for the instance portion and the `fn.ABS()` SQL function for the class portion.

```
class Interval(Model):
    start = IntegerField()
    end = IntegerField()

    @hybrid_property
    def length(self):
        return self.end - self.start

    @hybrid_property
    def radius(self):
        return abs(self.length) / 2

    @radius.expression
    def radius(cls):
        return fn.ABS(cls.length) / 2
```

What is neat is that both the `radius` implementations refer to the `length` hybrid attribute! When accessed via an `Interval` instance, the radius calculation will be executed in Python. When invoked via an `Interval` class, we

will get the appropriate SQL.

Example:

```
query = Interval.select().where(Interval.radius < 3)
```

This query is equivalent to the following SQL:

```
SELECT "t1"."id", "t1"."start", "t1"."end"
FROM "interval" AS t1
WHERE ((abs("t1"."end" - "t1"."start") / 2) < 3)
```

Pretty neat, right? Thanks for the cool idea, SQLAlchemy!

Hybrid API

class hybrid_method(*func*[, *expr=None*])

Method decorator that allows the definition of a Python object method with both instance-level and class-level behavior.

Example:

```
class Interval(Model):
    start = IntegerField()
    end = IntegerField()

    @hybrid_method
    def contains(self, point):
        return (self.start <= point) & (point < self.end)
```

When called with an `Interval` instance, the `contains` method will behave as you would expect. When called as a classmethod, though, a SQL expression will be generated:

```
query = Interval.select().where(Interval.contains(2))
```

Would generate the following SQL:

```
SELECT "t1"."id", "t1"."start", "t1"."end"
FROM "interval" AS t1
WHERE (("t1"."start" <= 2) AND (2 < "t1"."end"))
```

expression(*expr*)

Method decorator for specifying the SQL-expression producing method.

class hybrid_property(*fget*[, *fset=None*[, *fdel=None*[, *expr=None*]]])

Method decorator that allows the definition of a Python object property with both instance-level and class-level behavior.

Examples:

```
class Interval(Model):
    start = IntegerField()
    end = IntegerField()

    @hybrid_property
    def length(self):
        return self.end - self.start
```

(continues on next page)

(continued from previous page)

```
    @hybrid_property
    def radius(self):
        return abs(self.length) / 2

    @radius.expression
    def radius(cls):
        return fn.ABS(cls.length) / 2
```

When accessed on an `Interval` instance, the `length` and `radius` properties will behave as you would expect. When accessed as class attributes, though, a SQL expression will be generated instead:

```
query = (Interval
         .select()
         .where(
             (Interval.length > 6) &
             (Interval.radius >= 3)))
```

Would generate the following SQL:

```
SELECT "t1"."id", "t1"."start", "t1"."end"
FROM "interval" AS t1
WHERE (
    (("t1"."end" - "t1"."start") > 6) AND
    ((abs("t1"."end" - "t1"."start") / 2) >= 3)
)
```

1.13.12 Key/Value Store

The `playhouse.kv` module contains the implementation of a persistent dictionary.

class KeyValue([*key_field=None*[, *value_field=None*[, *ordered=False*[, *database=None*[, *table_name='keyvalue'*]]]]])

Parameters

- **key_field** (`Field`) – field to use for key. Defaults to *CharField*. **Must have** `primary_key=True`.
- **value_field** (`Field`) – field to use for value. Defaults to *PickleField*.
- **ordered** (*bool*) – data should be returned in key-sorted order.
- **database** (`Database`) – database where key/value data is stored. If not specified, an in-memory SQLite database will be used.
- **table_name** (*str*) – table name for data storage.

Dictionary-like API for storing key/value data. Like dictionaries, supports the expected APIs, but also has the added capability of accepting expressions for getting, setting and deleting items.

Table is created automatically (if it doesn't exist) when the `KeyValue` is instantiated.

Uses efficient upsert implementation for setting and updating/overwriting key/value pairs.

Basic examples:

```
# Create a key/value store, which uses an in-memory SQLite database
# for data storage.
KV = KeyValue()

# Set (or overwrite) the value for "k1".
KV['k1'] = 'v1'

# Set (or update) multiple keys at once (uses an efficient upsert).
KV.update(k2='v2', k3='v3')

# Getting values works as you'd expect.
assert KV['k2'] == 'v2'

# We can also do this:
for value in KV[KV.key > 'k1']:
    print(value)

# 'v2'
# 'v3'

# Update multiple values at once using expression:
KV[KV.key > 'k1'] = 'vx'

# What's stored in the KV?
print(dict(KV))

# {'k1': 'v1', 'k2': 'vx', 'k3': 'vx'}

# Delete a single item.
del KV['k2']

# How many items are stored in the KV?
print(len(KV))
# 2

# Delete items that match the given condition.
del KV[KV.key > 'k1']
```

__contains__(*expr*)

Parameters **expr** – a single key or an expression

Returns Boolean whether key/expression exists.

Example:

```
>>> kv = KeyValue()
>>> kv.update(k1='v1', k2='v2')

>>> 'k1' in kv
True
>>> 'kx' in kv
False

>>> (KV.key < 'k2') in KV
True
>>> (KV.key > 'k2') in KV
False
```

__len__()

Returns Count of items stored.

__getitem__(*expr*)

Parameters **expr** – a single key or an expression.

Returns value(s) corresponding to key/expression.

Raises `KeyError` if single key given and not found.

Examples:

```
>>> KV = KeyValue()
>>> KV.update(k1='v1', k2='v2', k3='v3')

>>> KV['k1']
'v1'
>>> KV['kx']
KeyError: "kx" not found

>>> KV[KV.key > 'k1']
['v2', 'v3']
>>> KV[KV.key < 'k1']
[]
```

__setitem__(*expr*, *value*)

Parameters

- **expr** – a single key or an expression.
- **value** – value to set for key(s)

Set value for the given key. If `expr` is an expression, then any keys matching the expression will have their value updated.

Example:

```
>>> KV = KeyValue()
>>> KV.update(k1='v1', k2='v2', k3='v3')

>>> KV['k1'] = 'v1-x'
>>> print(KV['k1'])
'v1-x'

>>> KV[KV.key >= 'k2'] = 'v99'
>>> dict(KV)
{'k1': 'v1-x', 'k2': 'v99', 'k3': 'v99'}
```

__delitem__(*expr*)

Parameters **expr** – a single key or an expression.

Delete the given key. If an expression is given, delete all keys that match the expression.

Example:

```
>>> KV = KeyValue()
>>> KV.update(k1=1, k2=2, k3=3)

>>> del KV['k1']  # Deletes "k1".
```

(continues on next page)

(continued from previous page)

```
>>> del KV['k1']
KeyError: "k1" does not exist

>>> del KV[KV.key > 'k2']  # Deletes "k3".
>>> del KV[KV.key > 'k99']  # Nothing deleted, no keys match.
```

keys()

Returns an iterable of all keys in the table.

values()

Returns an iterable of all values in the table.

items()

Returns an iterable of all key/value pairs in the table.

update(*[__data=None[, **mapping]]*)

Efficiently bulk-insert or replace the given key/value pairs.

Example:

```
>>> KV = KeyValue()
>>> KV.update(k1=1, k2=2)  # Sets 'k1'=1, 'k2'=2.

>>> dict(KV)
{'k1': 1, 'k2': 2}

>>> KV.update(k2=22, k3=3)  # Updates 'k2'->22, sets 'k3'=3.

>>> dict(KV)
{'k1': 1, 'k2': 22, 'k3': 3}

>>> KV.update({'k2': -2, 'k4': 4})  # Also can pass a dictionary.

>>> dict(KV)
{'k1': 1, 'k2': -2, 'k3': 3, 'k4': 4}
```

get(*expr[, default=None]*)

Parameters

- **expr** – a single key or an expression.
- **default** – default value if key not found.

Returns value of given key/expr or default if single key not found.

Get the value at the given key. If the key does not exist, the default value is returned, unless the key is an expression in which case an empty list will be returned.

pop(*expr[, default=Sentinel]*)

Parameters

- **expr** – a single key or an expression.
- **default** – default value if key does not exist.

Returns value of given key/expr or default if single key not found.

Get value and delete the given key. If the key does not exist, the default value is returned, unless the key is an expression in which case an empty list is returned.

clear()
Remove all items from the key-value table.

1.13.13 Shortcuts

This module contains helper functions for expressing things that would otherwise be somewhat verbose or cumbersome using peewee's APIs. There are also helpers for serializing models to dictionaries and vice-versa.

model_to_dict(*model*[, *recurse=True*[, *backrefs=False*[, *only=None*[, *exclude=None*[, *extra_attrs=None*[, *fields_from_query=None*[, *max_depth=None*[, *manytomany=False*]]]]]]]])

Parameters

- **recurse** (*bool*) – Whether foreign-keys should be recursed.
- **backrefs** (*bool*) – Whether lists of related objects should be recursed.
- **only** – A list (or set) of field instances which should be included in the result dictionary.
- **exclude** – A list (or set) of field instances which should be excluded from the result dictionary.
- **extra_attrs** – A list of attribute or method names on the instance which should be included in the dictionary.
- **fields_from_query** (Select) – The *SelectQuery* that created this model instance. Only the fields and values explicitly selected by the query will be serialized.
- **max_depth** (*int*) – Maximum depth when recursing.
- **manytomany** (*bool*) – Process many-to-many fields.

Convert a model instance (and optionally any related instances) to a dictionary.

Examples:

```
>>> user = User.create(username='charlie')
>>> model_to_dict(user)
{'id': 1, 'username': 'charlie'}

>>> model_to_dict(user, backrefs=True)
{'id': 1, 'tweets': [], 'username': 'charlie'}

>>> t1 = Tweet.create(user=user, message='tweet-1')
>>> t2 = Tweet.create(user=user, message='tweet-2')
>>> model_to_dict(user, backrefs=True)
{
  'id': 1,
  'tweets': [
    {'id': 1, 'message': 'tweet-1'},
    {'id': 2, 'message': 'tweet-2'},
  ],
  'username': 'charlie'
}

>>> model_to_dict(t1)
{
  'id': 1,
  'message': 'tweet-1',
  'user': {
```

(continues on next page)

(continued from previous page)

```
    'id': 1,
    'username': 'charlie'
  }
}

>>> model_to_dict(t2, recurse=False)
{'id': 1, 'message': 'tweet-2', 'user': 1}
```

The implementation of `model_to_dict` is fairly complex, owing to the various usages it attempts to support. If you have a special usage, I strongly advise that you do **not** attempt to shoe-horn some crazy combination of parameters into this function. Just write a simple function that accomplishes exactly what you're attempting to do.

dict_to_model(*model_class*, *data*[, *ignore_unknown=False*])

Parameters

- **model_class** (Model) – The model class to construct.
- **data** (*dict*) – A dictionary of data. Foreign keys can be included as nested dictionaries, and back-references as lists of dictionaries.
- **ignore_unknown** (*bool*) – Whether to allow unrecognized (non-field) attributes.

Convert a dictionary of data to a model instance, creating related instances where appropriate.

Examples:

```
>>> user_data = {'id': 1, 'username': 'charlie'}
>>> user = dict_to_model(User, user_data)
>>> user
<__main__.User at 0x7fea8fa4d490>

>>> user.username
'charlie'

>>> note_data = {'id': 2, 'text': 'note text', 'user': user_data}
>>> note = dict_to_model(Note, note_data)
>>> note.text
'note text'
>>> note.user.username
'charlie'

>>> user_with_notes = {
...     'id': 1,
...     'username': 'charlie',
...     'notes': [{'id': 1, 'text': 'note-1'}, {'id': 2, 'text': 'note-2'}]}
>>> user = dict_to_model(User, user_with_notes)
>>> user.notes[0].text
'note-1'
>>> user.notes[0].user.username
'charlie'
```

update_model_from_dict(*instance*, *data*[, *ignore_unknown=False*])

Parameters

- **instance** (Model) – The model instance to update.

- **data** (*dict*) – A dictionary of data. Foreign keys can be included as nested dictionaries, and back-references as lists of dictionaries.
- **ignore_unknown** (*bool*) – Whether to allow unrecognized (non-field) attributes.

Update a model instance with the given data dictionary.

1.13.14 Signal support

Models with hooks for signals (a-la django) are provided in `playhouse.signals`. To use the signals, you will need all of your project's models to be a subclass of `playhouse.signals.Model`, which overrides the necessary methods to provide support for the various signals.

```
from playhouse.signals import Model, post_save


class MyModel(Model):
    data = IntegerField()

@post_save(sender=MyModel)
def on_save_handler(model_class, instance, created):
    put_data_in_cache(instance.data)
```

Warning: For what I hope are obvious reasons, Peewee signals do not work when you use the *Model.insert()*, *Model.update()*, or *Model.delete()* methods. These methods generate queries that execute beyond the scope of the ORM, and the ORM does not know about which model instances might or might not be affected when the query executes.

Signals work by hooking into the higher-level peewee APIs like *Model.save()* and *Model.delete_instance()*, where the affected model instance is known ahead of time.

The following signals are provided:

pre_save Called immediately before an object is saved to the database. Provides an additional keyword argument `created`, indicating whether the model is being saved for the first time or updated.

post_save Called immediately after an object is saved to the database. Provides an additional keyword argument `created`, indicating whether the model is being saved for the first time or updated.

pre_delete Called immediately before an object is deleted from the database when *Model.delete_instance()* is used.

post_delete Called immediately after an object is deleted from the database when *Model.delete_instance()* is used.

pre_init Called when a model class is first instantiated

Connecting handlers

Whenever a signal is dispatched, it will call any handlers that have been registered. This allows totally separate code to respond to events like model save and delete.

The *Signal* class provides a *connect()* method, which takes a callback function and two optional parameters for "sender" and "name". If specified, the "sender" parameter should be a single model class and allows your callback to only receive signals from that one model class. The "name" parameter is used as a convenient alias in the event you wish to unregister your signal handler.

Example usage:

```
from playhouse.signals import *

def post_save_handler(sender, instance, created):
    print '%s was just saved' % instance

# our handler will only be called when we save instances of SomeModel
post_save.connect(post_save_handler, sender=SomeModel)
```

All signal handlers accept as their first two arguments `sender` and `instance`, where `sender` is the model class and `instance` is the actual model being acted upon.

If you'd like, you can also use a decorator to connect signal handlers. This is functionally equivalent to the above example:

```
@post_save(sender=SomeModel)
def post_save_handler(sender, instance, created):
    print '%s was just saved' % instance
```

Signal API

class Signal

Stores a list of receivers (callbacks) and calls them when the "send" method is invoked.

connect(*receiver*[, *sender=None*[, *name=None*]])

Parameters

- **receiver** (*callable*) – a callable that takes at least two parameters, a "sender", which is the Model subclass that triggered the signal, and an "instance", which is the actual model instance.
- **sender** (Model) – if specified, only instances of this model class will trigger the receiver callback.
- **name** (*string*) – a short alias

Add the receiver to the internal list of receivers, which will be called whenever the signal is sent.

```
from playhouse.signals import post_save
from project.handlers import cache_buster

post_save.connect(cache_buster, name='project.cache_buster')
```

disconnect([*receiver=None*[, *name=None*]])

Parameters

- **receiver** (*callable*) – the callback to disconnect
- **name** (*string*) – a short alias

Disconnect the given receiver (or the receiver with the given name alias) so that it no longer is called. Either the receiver or the name must be provided.

```
post_save.disconnect(name='project.cache_buster')
```

send(*instance*, **args*, ***kwargs*)

Parameters **instance** – a model instance

Iterates over the receivers and will call them in the order in which they were connected. If the receiver specified a sender, it will only be called if the instance is an instance of the sender.

1.13.15 pwiz, a model generator

`pwiz` is a little script that ships with peewee and is capable of introspecting an existing database and generating model code suitable for interacting with the underlying data. If you have a database already, pwiz can give you a nice boost by generating skeleton code with correct column affinities and foreign keys.

If you install peewee using `setup.py install`, pwiz will be installed as a "script" and you can just run:

```
python -m pwiz -e postgresql -u postgres my_postgres_db
```

This will print a bunch of models to standard output. So you can do this:

```
python -m pwiz -e postgresql my_postgres_db > mymodels.py
python # <-- fire up an interactive shell
```

```
>>> from mymodels import Blog, Entry, Tag, Whatever
>>> print [blog.name for blog in Blog.select()]
```

Command-line options

pwiz accepts the following command-line options:

Option	Meaning	Example
-h	show help	
-e	database backend	-e mysql
-H	host to connect to	-H remote.db.server
-p	port to connect on	-p 9001
-u	database user	-u postgres
-P	database password	-P (will be prompted for password)
-s	schema	-s public
-t	tables to generate	-t tweet,users,relationships
-v	generate models for VIEWs	(no argument)
-i	add info metadata to generated file	(no argument)
-o	table column order is preserved	(no argument)

The following are valid parameters for the `engine` (`-e`):

- sqlite
- mysql
- postgresql

Warning: If a password is required to access your database, you will be prompted to enter it using a secure prompt.

The password will be included in the output. Specifically, at the top of the file a *Database* will be defined along with any required parameters – including the password.

pwiz examples

Examples of introspecting various databases:

```
# Introspect a Sqlite database.
python -m pwiz -e sqlite path/to/sqlite_database.db

# Introspect a MySQL database, logging in as root. You will be prompted
# for a password ("-P").
python -m pwiz -e mysql -u root -P mysql_db_name

# Introspect a Postgresql database on a remote server.
python -m pwiz -e postgres -u postgres -H 10.1.0.3 pg_db_name
```

Full example:

```
$ sqlite3 example.db << EOM
CREATE TABLE "user" ("id" INTEGER NOT NULL PRIMARY KEY, "username" TEXT NOT NULL);
CREATE TABLE "tweet" (
    "id" INTEGER NOT NULL PRIMARY KEY,
    "content" TEXT NOT NULL,
    "timestamp" DATETIME NOT NULL,
    "user_id" INTEGER NOT NULL,
    FOREIGN KEY ("user_id") REFERENCES "user" ("id"));
CREATE UNIQUE INDEX "user_username" ON "user" ("username");
EOM

$ python -m pwiz -e sqlite example.db
```

Produces the following output:

```
from peewee import *

database = SqliteDatabase('example.db', **{})

class UnknownField(object):
    def __init__(self, *_, **__): pass

class BaseModel(Model):
    class Meta:
        database = database

class User(BaseModel):
    username = TextField(unique=True)

    class Meta:
        table_name = 'user'

class Tweet(BaseModel):
    content = TextField()
    timestamp = DateTimeField()
    user = ForeignKeyField(column_name='user_id', field='id', model=User)

    class Meta:
        table_name = 'tweet'
```

Observations:

- The foreign-key `Tweet.user_id` is detected and mapped correctly.

- The `User.username` UNIQUE constraint is detected.
- Each model explicitly declares its table name, even in cases where it is not necessary (as Peewee would automatically translate the class name into the appropriate table name).
- All the parameters of the *`ForeignKeyField`* are explicitly declared, even though they follow the conventions Peewee uses by default.

Note: The `UnknownField` is a placeholder that is used in the event your schema contains a column declaration that Peewee doesn't know how to map to a field class.

1.13.16 Schema Migrations

Peewee now supports schema migrations, with well-tested support for Postgresql, SQLite and MySQL. Unlike other schema migration tools, peewee's migrations do not handle introspection and database "versioning". Rather, peewee provides a number of helper functions for generating and running schema-altering statements. This engine provides the basis on which a more sophisticated tool could some day be built.

Migrations can be written as simple python scripts and executed from the command-line. Since the migrations only depend on your applications *`Database`* object, it should be easy to manage changing your model definitions and maintaining a set of migration scripts without introducing dependencies.

Example usage

Begin by importing the helpers from the *migrate* module:

```
from playhouse.migrate import *
```

Instantiate a `migrator`. The *`SchemaMigrator`* class is responsible for generating schema altering operations, which can then be run sequentially by the `migrate()` helper.

```
# Postgres example:
my_db = PostgresqlDatabase(...)
migrator = PostgresqlMigrator(my_db)

# SQLite example:
my_db = SqliteDatabase('my_database.db')
migrator = SqliteMigrator(my_db)
```

Use `migrate()` to execute one or more operations:

```
title_field = CharField(default='')
status_field = IntegerField(null=True)

migrate(
    migrator.add_column('some_table', 'title', title_field),
    migrator.add_column('some_table', 'status', status_field),
    migrator.drop_column('some_table', 'old_column'),
)
```

Warning: Migrations are not run inside a transaction. If you wish the migration to run in a transaction you will need to wrap the call to *migrate* in a *`atomic()`* context-manager, e.g.

```
with my_db.atomic():
    migrate(...)
```

Supported Operations

Add new field(s) to an existing model:

```
# Create your field instances. For non-null fields you must specify a
# default value.
pubdate_field = DateTimeField(null=True)
comment_field = TextField(default='')

# Run the migration, specifying the database table, field name and field.
migrate(
    migrator.add_column('comment_tbl', 'pub_date', pubdate_field),
    migrator.add_column('comment_tbl', 'comment', comment_field),
)
```

Renaming a field:

```
# Specify the table, original name of the column, and its new name.
migrate(
    migrator.rename_column('story', 'pub_date', 'publish_date'),
    migrator.rename_column('story', 'mod_date', 'modified_date'),
)
```

Dropping a field:

```
migrate(
    migrator.drop_column('story', 'some_old_field'),
)
```

Making a field nullable or not nullable:

```
# Note that when making a field not null that field must not have any
# NULL values present.
migrate(
    # Make `pub_date` allow NULL values.
    migrator.drop_not_null('story', 'pub_date'),

    # Prevent `modified_date` from containing NULL values.
    migrator.add_not_null('story', 'modified_date'),
)
```

Altering a field's data-type:

```
# Change a VARCHAR(50) field to a TEXT field.
migrate(
    migrator.alter_column_type('person', 'email', TextField())
)
```

Renaming a table:

```
migrate(
    migrator.rename_table('story', 'stories_tbl'),
)
```

Adding an index:

```
# Specify the table, column names, and whether the index should be
# UNIQUE or not.
migrate(
    # Create an index on the `pub_date` column.
    migrator.add_index('story', ('pub_date',), False),

    # Create a multi-column index on the `pub_date` and `status` fields.
    migrator.add_index('story', ('pub_date', 'status'), False),

    # Create a unique index on the category and title fields.
    migrator.add_index('story', ('category_id', 'title'), True),
)
```

Dropping an index:

```
# Specify the index name.
migrate(migrator.drop_index('story', 'story_pub_date_status'))
```

Adding or dropping table constraints:

```
# Add a CHECK() constraint to enforce the price cannot be negative.
migrate(migrator.add_constraint(
    'products',
    'price_check',
    Check('price >= 0')))

# Remove the price check constraint.
migrate(migrator.drop_constraint('products', 'price_check'))

# Add a UNIQUE constraint on the first and last names.
migrate(migrator.add_unique('person', 'first_name', 'last_name'))
```

Migrations API

migrate(**operations*)

Execute one or more schema altering operations.

Usage:

```
migrate(
    migrator.add_column('some_table', 'new_column', CharField(default='')),
    migrator.create_index('some_table', ('new_column',)),
)
```

class SchemaMigrator(*database*)

Parameters **database** – a *Database* instance.

The *SchemaMigrator* is responsible for generating schema-altering statements.

add_column(*table*, *column_name*, *field*)

Parameters

- **table** (*str*) – Name of the table to add column to.
- **column_name** (*str*) – Name of the new column.
- **field** (Field) – A *Field* instance.

Add a new column to the provided table. The `field` provided will be used to generate the appropriate column definition.

Note: If the field is not nullable it must specify a default value.

Note: For non-null fields, the field will initially be added as a null field, then an `UPDATE` statement will be executed to populate the column with the default value. Finally, the column will be marked as not null.

---

drop_column(*table*, *column_name*[, *cascade=True*])

Parameters

- **table** (*str*) – Name of the table to drop column from.
- **column_name** (*str*) – Name of the column to drop.
- **cascade** (*bool*) – Whether the column should be dropped with *CASCADE*.

rename_column(*table*, *old_name*, *new_name*)

Parameters

- **table** (*str*) – Name of the table containing column to rename.
- **old_name** (*str*) – Current name of the column.
- **new_name** (*str*) – New name for the column.

add_not_null(*table*, *column*)

Parameters

- **table** (*str*) – Name of table containing column.
- **column** (*str*) – Name of the column to make not nullable.

drop_not_null(*table*, *column*)

Parameters

- **table** (*str*) – Name of table containing column.
- **column** (*str*) – Name of the column to make nullable.

alter_column_type(*table*, *column*, *field*[, *cast=None*])

Parameters

- **table** (*str*) – Name of the table.
- **column_name** (*str*) – Name of the column to modify.
- **field** (Field) – *Field* instance representing new data type.
- **cast** – (postgres-only) specify a cast expression if the data-types are incompatible, e.g. `column_name::int`. Can be provided as either a string or a *Cast* instance.

Alter the data-type of a column. This method should be used with care, as using incompatible types may not be well-supported by your database.

rename_table(*old_name*, *new_name*)

Parameters

- **old_name** (*str*) – Current name of the table.
- **new_name** (*str*) – New name for the table.

add_index(*table*, *columns*[, *unique=False*[, *using=None*]])

Parameters

- **table** (*str*) – Name of table on which to create the index.
- **columns** (*list*) – List of columns which should be indexed.
- **unique** (*bool*) – Whether the new index should specify a unique constraint.
- **using** (*str*) – Index type (where supported), e.g. GiST or GIN.

drop_index(*table*, *index_name*)

Parameters

- **table** (*str*) – Name of the table containing the index to be dropped.
- **index_name** (*str*) – Name of the index to be dropped.

add_constraint(*table*, *name*, *constraint*)

Parameters

- **table** (*str*) – Table to add constraint to.
- **name** (*str*) – Name used to identify the constraint.
- **constraint** – either a *Check()* constraint or for adding an arbitrary constraint use *SQL*.

drop_constraint(*table*, *name*)

Parameters

- **table** (*str*) – Table to drop constraint from.
- **name** (*str*) – Name of constraint to drop.

add_unique(*table*, **column_names*)

Parameters

- **table** (*str*) – Table to add constraint to.
- **column_names** (*str*) – One or more columns for UNIQUE constraint.

class PostgresqlMigrator(*database*)

Generate migrations for Postgresql databases.

set_search_path(*schema_name*)

Parameters **schema_name** (*str*) – Schema to use.

Set the search path (schema) for the subsequent operations.

class SqliteMigrator(*database*)
Generate migrations for SQLite databases.

SQLite has limited support for `ALTER TABLE` queries, so the following operations are currently not supported for SQLite:

- `add_constraint`
- `drop_constraint`
- `add_unique`

class MySQLMigrator(*database*)
Generate migrations for MySQL databases.

1.13.17 Reflection

The reflection module contains helpers for introspecting existing databases. This module is used internally by several other modules in the playhouse, including *DataSet* and *pwiz, a model generator*.

generate_models(*database*[, *schema=None*[, ***options*]])

Parameters

- **database** (Database) – database instance to introspect.
- **schema** (*str*) – optional schema to introspect.
- **options** – arbitrary options, see *Introspector.generate_models()* for details.

Returns a `dict` mapping table names to model classes.

Generate models for the tables in the given database. For an example of how to use this function, see the section *Using Peewee Interactively*.

Example:

```
>>> from peewee import *
>>> from playhouse.reflection import generate_models
>>> db = PostgresqlDatabase('my_app')
>>> models = generate_models(db)
>>> list(models.keys())
['account', 'customer', 'order', 'orderitem', 'product']

>>> globals().update(models)  # Inject models into namespace.
>>> for cust in customer.select():  # Query using generated model.
...     print(cust.name)
...

Huey Kitty
Mickey Dog
```

print_model(*model*)

Parameters **model** (Model) – model class to print

Returns no return value

Print a user-friendly description of a model class, useful for debugging or interactive use. Currently this prints the table name, and all fields along with their data-types. The *Using Peewee Interactively* section contains an example.

Example output:

```
>>> from playhouse.reflection import print_model
>>> print_model(User)
user
  id AUTO PK
  email TEXT
  name TEXT
  dob DATE

index(es)
  email UNIQUE

>>> print_model(Tweet)
tweet
  id AUTO PK
  user INT FK: User.id
  title TEXT
  content TEXT
  timestamp DATETIME
  is_published BOOL

index(es)
  user_id
  is_published, timestamp
```

print_table_sql(*model*)

Parameters **model** (Model) – model to print

Returns no return value

Prints the SQL `CREATE TABLE` for the given model class, which may be useful for debugging or interactive use. See the *Using Peewee Interactively* section for example usage. Note that indexes and constraints are not included in the output of this function.

Example output:

```
>>> from playhouse.reflection import print_table_sql
>>> print_table_sql(User)
CREATE TABLE IF NOT EXISTS "user" (
  "id" INTEGER NOT NULL PRIMARY KEY,
  "email" TEXT NOT NULL,
  "name" TEXT NOT NULL,
  "dob" DATE NOT NULL
)

>>> print_table_sql(Tweet)
CREATE TABLE IF NOT EXISTS "tweet" (
  "id" INTEGER NOT NULL PRIMARY KEY,
  "user_id" INTEGER NOT NULL,
  "title" TEXT NOT NULL,
  "content" TEXT NOT NULL,
  "timestamp" DATETIME NOT NULL,
  "is_published" INTEGER NOT NULL,
  FOREIGN KEY ("user_id") REFERENCES "user" ("id")
)
```

class Introspector(*metadata*[, *schema=None*])

Metadata can be extracted from a database by instantiating an *Introspector*. Rather than instantiating this class directly, it is recommended to use the factory method *from_database()*.

classmethod from_database(*database*[, *schema=None*])

Parameters

- **database** – a *Database* instance.
- **schema** (*str*) – an optional schema (supported by some databases).

Creates an *Introspector* instance suitable for use with the given database.

Usage:

```
db = SqliteDatabase('my_app.db')
introspector = Introspector.from_database(db)
models = introspector.generate_models()

# User and Tweet (assumed to exist in the database) are
# peewee Model classes generated from the database schema.
User = models['user']
Tweet = models['tweet']
```

generate_models([*skip_invalid=False*[, *table_names=None*[, *literal_column_names=False*[, *bare_fields=False*[, *include_views=False*]]]]])

Parameters

- **skip_invalid** (*bool*) – Skip tables whose names are invalid python identifiers.
- **table_names** (*list*) – List of table names to generate. If unspecified, models are generated for all tables.
- **literal_column_names** (*bool*) – Use column-names as-is. By default, column names are "python-ized", i.e. mixed-case becomes lower-case.
- **bare_fields** – **SQLite-only**. Do not specify data-types for introspected columns.
- **include_views** – generate models for VIEWs as well.

Returns A dictionary mapping table-names to model classes.

Introspect the database, reading in the tables, columns, and foreign key constraints, then generate a dictionary mapping each database table to a dynamically-generated *Model* class.

1.13.18 Database URL

This module contains a helper function to generate a database connection from a URL connection string.

connect(*url*, ***connect_params*)

Create a *Database* instance from the given connection URL.

Examples:

- *sqlite:///my_database.db* will create a *SqliteDatabase* instance for the file `my_database.db` in the current directory.
- *sqlite:///:memory:* will create an in-memory *SqliteDatabase* instance.
- *postgresql://postgres:my_password@localhost:5432/my_database* will create a *PostgresqlDatabase* instance. A username and password are provided, as well as the host and port to connect to.
- *mysql://user:passwd@ip:port/my_db* will create a *MySQLDatabase* instance for the local MySQL database *my_db*.

- *mysql+pool://user:passwd@ip:port/my_db?max_connections=20&stale_timeout=300* will create a *PooledMySQLDatabase* instance for the local MySQL database *my_db* with max_connections set to 20 and a stale_timeout setting of 300 seconds.

Supported schemes:

- `apsw`: *APSWDatabase*
- `mysql`: *MySQLDatabase*
- `mysql+pool`: *PooledMySQLDatabase*
- `postgres`: *PostgresqlDatabase*
- `postgres+pool`: *PooledPostgresqlDatabase*
- `postgresext`: *PostgresqlExtDatabase*
- `postgresext+pool`: *PooledPostgresqlExtDatabase*
- `sqlite`: *SqliteDatabase*
- `sqliteext`: *SqliteExtDatabase*
- `sqlite+pool`: *PooledSqliteDatabase*
- `sqliteext+pool`: *PooledSqliteExtDatabase*

Usage:

```
import os
from playhouse.db_url import connect

# Connect to the database URL defined in the environment, falling
# back to a local Sqlite database if no database URL is specified.
db = connect(os.environ.get('DATABASE') or 'sqlite:///default.db')
```

parse(*url*)

Parse the information in the given URL into a dictionary containing `database`, `host`, `port`, `user` and/or `password`. Additional connection arguments can be passed in the URL query string.

If you are using a custom database class, you can use the `parse()` function to extract information from a URL which can then be passed in to your database object.

register_database(*db_class*, **names*)

Parameters

- **db_class** – A subclass of *Database*.
- **names** – A list of names to use as the scheme in the URL, e.g. 'sqlite' or 'firebird'

Register additional database class under the specified names. This function can be used to extend the `connect()` function to support additional schemes. Suppose you have a custom database class for `Firebird` named `FirebirdDatabase`.

```
from playhouse.db_url import connect, register_database

register_database(FirebirdDatabase, 'firebird')
db = connect('firebird://my-firebird-db')
```

1.13.19 Connection pool

The `pool` module contains a number of *Database* classes that provide connection pooling for PostgreSQL, MySQL and SQLite databases. The pool works by overriding the methods on the *Database* class that open and close connections to the backend. The pool can specify a timeout after which connections are recycled, as well as an upper bound on the number of open connections.

In a multi-threaded application, up to *max_connections* will be opened. Each thread (or, if using gevent, greenlet) will have its own connection.

In a single-threaded application, only one connection will be created. It will be continually recycled until either it exceeds the stale timeout or is closed explicitly (using *.manual_close()*).

By default, all your application needs to do is ensure that connections are closed when you are finished with them, and they will be returned to the pool. For web applications, this typically means that at the beginning of a request, you will open a connection, and when you return a response, you will close the connection.

Simple Postgres pool example code:

```
# Use the special postgresql extensions.
from playhouse.pool import PooledPostgresqlExtDatabase

db = PooledPostgresqlExtDatabase(
    'my_app',
    max_connections=32,
    stale_timeout=300,  # 5 minutes.
    user='postgres')

class BaseModel(Model):
    class Meta:
        database = db
```

That's it! If you would like finer-grained control over the pool of connections, check out the advanced_connection_management section.

Pool APIs

class PooledDatabase(*database*[, *max_connections=20*[, *stale_timeout=None*[, *timeout=None*[, ***kwargs*]]]])

Parameters

- **database** (*str*) – The name of the database or database file.
- **max_connections** (*int*) – Maximum number of connections. Provide `None` for unlimited.
- **stale_timeout** (*int*) – Number of seconds to allow connections to be used.
- **timeout** (*int*) – Number of seconds to block when pool is full. By default peewee does not block when the pool is full but simply throws an exception. To block indefinitely set this value to `0`.
- **kwargs** – Arbitrary keyword arguments passed to database class.

Mixin class intended to be used with a subclass of *Database*.

Note: Connections will not be closed exactly when they exceed their *stale_timeout*. Instead, stale connections are only closed when a new connection is requested.

Note: If the number of open connections exceeds *max_connections*, a *ValueError* will be raised.

manual_close()
Close the currently-open connection without returning it to the pool.

close_idle()
Close all idle connections. This does not include any connections that are currently in-use – only those that were previously created but have since been returned back to the pool.

close_stale([*age=600*])

Parameters **age** (*int*) – Age at which a connection should be considered stale.

Returns Number of connections closed.

Close connections which are in-use but exceed the given age. **Use caution when calling this method!**

close_all()
Close all connections. This includes any connections that may be in use at the time. **Use caution when calling this method!**

class PooledPostgresqlDatabase
Subclass of *PostgresqlDatabase* that mixes in the *PooledDatabase* helper.

class PooledPostgresqlExtDatabase
Subclass of *PostgresqlExtDatabase* that mixes in the *PooledDatabase* helper. The *PostgresqlExtDatabase* is a part of the *Postgresql Extensions* module and provides support for many Postgres-specific features.

class PooledMySQLDatabase
Subclass of *MySQLDatabase* that mixes in the *PooledDatabase* helper.

class PooledSqliteDatabase
Persistent connections for SQLite apps.

class PooledSqliteExtDatabase
Persistent connections for SQLite apps, using the *SQLite Extensions* advanced database driver *SqliteExtDatabase*.

1.13.20 Test Utils

Contains utilities helpful when testing peewee projects.

class count_queries([*only_select=False*])
Context manager that will count the number of queries executed within the context.

Parameters **only_select** (*bool*) – Only count *SELECT* queries.

```
with count_queries() as counter:
    huey = User.get(User.username == 'huey')
    huey_tweets = [tweet.message for tweet in huey.tweets]

assert counter.count == 2
```

count
The number of queries executed.

get_queries()
Return a list of 2-tuples consisting of the SQL query and a list of parameters.

assert_query_count(*expected*[, *only_select=False*])
Function or method decorator that will raise an `AssertionError` if the number of queries executed in the decorated function does not equal the expected number.

```
class TestMyApp(unittest.TestCase):
    @assert_query_count(1)
    def test_get_popular_blogs(self):
        popular_blogs = Blog.get_popular()
        self.assertEqual(
            [blog.title for blog in popular_blogs],
            ["Peewee's Playhouse!", "All About Huey", "Mickey's Adventures"])
```

This function can also be used as a context manager:

```
class TestMyApp(unittest.TestCase):
    def test_expensive_operation(self):
        with assert_query_count(1):
            perform_expensive_operation()
```

1.13.21 Flask Utils

The `playhouse.flask_utils` module contains several helpers for integrating peewee with the Flask web framework.

Database Wrapper

The `FlaskDB` class is a wrapper for configuring and referencing a Peewee database from within a Flask application. Don't let its name fool you: it is **not the same thing as a peewee database**. `FlaskDB` is designed to remove the following boilerplate from your flask app:

- Dynamically create a Peewee database instance based on app config data.
- Create a base class from which all your application's models will descend.
- Register hooks at the start and end of a request to handle opening and closing a database connection.

Basic usage:

```
import datetime
from flask import Flask
from peewee import *
from playhouse.flask_utils import FlaskDB

DATABASE = 'postgresql://postgres:password@localhost:5432/my_database'

app = Flask(__name__)
app.config.from_object(__name__)

db_wrapper = FlaskDB(app)

class User(db_wrapper.Model):
```

(continues on next page)

(continued from previous page)

```
    username = CharField(unique=True)

class Tweet(db_wrapper.Model):
    user = ForeignKeyField(User, backref='tweets')
    content = TextField()
    timestamp = DateTimeField(default=datetime.datetime.now)
```

The above code example will create and instantiate a peewee *PostgresqlDatabase* specified by the given database URL. Request hooks will be configured to establish a connection when a request is received, and automatically close the connection when the response is sent. Lastly, the `FlaskDB` class exposes a `FlaskDB.Model` property which can be used as a base for your application's models.

Here is how you can access the wrapped Peewee database instance that is configured for you by the `FlaskDB` wrapper:

```
# Obtain a reference to the Peewee database instance.
peewee_db = db_wrapper.database

@app.route('/transfer-funds/', methods=['POST'])
def transfer_funds():
    with peewee_db.atomic():
        # ...

    return jsonify({'transfer-id': xid})
```

Note: The actual peewee database can be accessed using the `FlaskDB.database` attribute.

Here is another way to configure a Peewee database using `FlaskDB`:

```
app = Flask(__name__)
db_wrapper = FlaskDB(app, 'sqlite:///my_app.db')
```

While the above examples show using a database URL, for more advanced usages you can specify a dictionary of configuration options, or simply pass in a peewee *Database* instance:

```
DATABASE = {
    'name': 'my_app_db',
    'engine': 'playhouse.pool.PooledPostgresqlDatabase',
    'user': 'postgres',
    'max_connections': 32,
    'stale_timeout': 600,
}

app = Flask(__name__)
app.config.from_object(__name__)

wrapper = FlaskDB(app)
pooled_postgres_db = wrapper.database
```

Using a peewee *Database* object:

```
peewee_db = PostgresqlExtDatabase('my_app')
app = Flask(__name__)
db_wrapper = FlaskDB(app, peewee_db)
```

Database with Application Factory

If you prefer to use the application factory pattern, the `FlaskDB` class implements an `init_app()` method.

Using as a factory:

```
db_wrapper = FlaskDB()

# Even though the database is not yet initialized, you can still use the
# `Model` property to create model classes.
class User(db_wrapper.Model):
    username = CharField(unique=True)


def create_app():
    app = Flask(__name__)
    app.config['DATABASE'] = 'sqlite:////home/code/apps/my-database.db'
    db_wrapper.init_app(app)
    return app
```

Query utilities

The `flask_utils` module provides several helpers for managing queries in your web app. Some common patterns include:

get_object_or_404(*query_or_model*, **query*)

Parameters

- **query_or_model** – Either a *Model* class or a pre-filtered *SelectQuery*.
- **query** – An arbitrarily complex peewee expression.

Retrieve the object matching the given query, or return a 404 not found response. A common use-case might be a detail page for a weblog. You want to either retrieve the post matching the given URL, or return a 404.

Example:

```
@app.route('/blog/<slug>/')
def post_detail(slug):
    public_posts = Post.select().where(Post.published == True)
    post = get_object_or_404(public_posts, (Post.slug == slug))
    return render_template('post_detail.html', post=post)
```

object_list(*template_name*, *query*[, *context_variable='object_list'*[, *paginate_by=20*[, *page_var='page'*[, *check_bounds=True*[, ***kwargs*]]]]])

Parameters

- **template_name** – The name of the template to render.
- **query** – A *SelectQuery* instance to paginate.
- **context_variable** – The context variable name to use for the paginated object list.
- **paginate_by** – Number of objects per-page.
- **page_var** – The name of the `GET` argument which contains the page.
- **check_bounds** – Whether to check that the given page is a valid page. If `check_bounds` is `True` and an invalid page is specified, then a 404 will be returned.

- **kwargs** – Arbitrary key/value pairs to pass into the template context.

Retrieve a paginated list of objects specified by the given query. The paginated object list will be dropped into the context using the given `context_variable`, as well as metadata about the current page and total number of pages, and finally any arbitrary context data passed as keyword-arguments.

The page is specified using the `page` `GET` argument, e.g. `/my-object-list/?page=3` would return the third page of objects.

Example:

```
@app.route('/blog/')
def post_index():
    public_posts = (Post
                    .select()
                    .where(Post.published == True)
                    .order_by(Post.timestamp.desc()))

    return object_list(
        'post_index.html',
        query=public_posts,
        context_variable='post_list',
        paginate_by=10)
```

The template will have the following context:

- `post_list`, which contains a list of up to 10 posts.
- `page`, which contains the current page based on the value of the `page` `GET` parameter.
- `pagination`, a *PaginatedQuery* instance.

class PaginatedQuery(*query_or_model*, *paginate_by*[, *page_var='page'*[, *check_bounds=False*]])

Parameters

- **query_or_model** – Either a *Model* or a *SelectQuery* instance containing the collection of records you wish to paginate.
- **paginate_by** – Number of objects per-page.
- **page_var** – The name of the `GET` argument which contains the page.
- **check_bounds** – Whether to check that the given page is a valid page. If `check_bounds` is `True` and an invalid page is specified, then a 404 will be returned.

Helper class to perform pagination based on `GET` arguments.

get_page()
Return the currently selected page, as indicated by the value of the `page_var` `GET` parameter. If no page is explicitly selected, then this method will return 1, indicating the first page.

get_page_count()
Return the total number of possible pages.

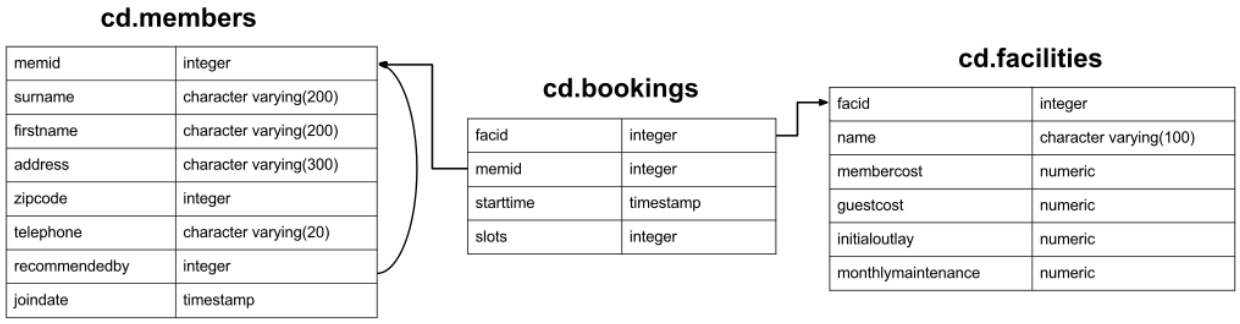
get_object_list()
Using the value of *get_page()*, return the page of objects requested by the user. The return value is a *SelectQuery* with the appropriate `LIMIT` and `OFFSET` clauses.

If `check_bounds` was set to `True` and the requested page contains no objects, then a 404 will be raised.

1.14 Query Examples

These query examples are taken from the site PostgreSQL Exercises. A sample data-set can be found on the getting started page.

Here is a visual representation of the schema used in these examples:



1.14.1 Model Definitions

To begin working with the data, we'll define the model classes that correspond to the tables in the diagram.

Note: In some cases we explicitly specify column names for a particular field. This is so our models are compatible with the database schema used for the postgres exercises.

```
from functools import partial
from peewee import *


db = PostgresqlDatabase('peewee_test')

class BaseModel(Model):
    class Meta:
        database = db

class Member(BaseModel):
    memid = AutoField()  # Auto-incrementing primary key.
    surname = CharField()
    firstname = CharField()
    address = CharField(max_length=300)
    zipcode = IntegerField()
    telephone = CharField()
    recommendedby = ForeignKeyField('self', backref='recommended',
                                    column_name='recommendedby', null=True)
    joindate = DateTimeField()

    class Meta:
        table_name = 'members'


# Conveniently declare decimal fields suitable for storing currency.
MoneyField = partial(DecimalField, decimal_places=2)
```

(continues on next page)

(continued from previous page)

```
class Facility(BaseModel):
    facid = AutoField()
    name = CharField()
    membercost = MoneyField()
    guestcost = MoneyField()
    initialoutlay = MoneyField()
    monthlymaintenance = MoneyField()

    class Meta:
        table_name = 'facilities'


class Booking(BaseModel):
    bookid = AutoField()
    facility = ForeignKeyField(Facility, column_name='facid')
    member = ForeignKeyField(Member, column_name='memid')
    starttime = DateTimeField()
    slots = IntegerField()

    class Meta:
        table_name = 'bookings'
```

1.14.2 Schema Creation

If you downloaded the SQL file from the PostgreSQL Exercises site, then you can load the data into a PostgreSQL database using the following commands:

```
createdb peewee_test
psql -U postgres -f clubdata.sql -d peewee_test -x -q
```

To create the schema using Peewee, without loading the sample data, you can run the following:

```
# Assumes you have created the database "peewee_test" already.
db.create_tables([Member, Facility, Booking])
```

1.14.3 Basic Exercises

This category deals with the basics of SQL. It covers select and where clauses, case expressions, unions, and a few other odds and ends.

Retrieve everything

Retrieve all information from facilities table.

```
SELECT * FROM facilities
```

```
# By default, when no fields are explicitly passed to select(), all fields
# will be selected.
query = Facility.select()
```

Retrieve specific columns from a table

Retrieve names of facilities and cost to members.

```
SELECT name, membercost FROM facilities;
```

```
query = Facility.select(Facility.name, Facility.membercost)

# To iterate:
for facility in query:
    print(facility.name)
```

Control which rows are retrieved

Retrieve list of facilities that have a cost to members.

```
SELECT * FROM facilities WHERE membercost > 0
```

```
query = Facility.select().where(Facility.membercost > 0)
```

Control which rows are retrieved - part 2

Retrieve list of facilities that have a cost to members, and that fee is less than 1/50th of the monthly maintenance cost. Return id, name, cost and monthly-maintenance.

```
SELECT facid, name, membercost, monthlymaintenance
FROM facilities
WHERE membercost > 0 AND membercost < (monthlymaintenance / 50)
```

```
query = (Facility
         .select(Facility.facid, Facility.name, Facility.membercost,
                 Facility.monthlymaintenance)
         .where(
             (Facility.membercost > 0) &
             (Facility.membercost < (Facility.monthlymaintenance / 50))))
```

Basic string searches

How can you produce a list of all facilities with the word 'Tennis' in their name?

```
SELECT * FROM facilities WHERE name ILIKE '%tennis%';
```

```
query = Facility.select().where(Facility.name.contains('tennis'))

# OR use the exponent operator. Note: you must include wildcards here:
query = Facility.select().where(Facility.name ** '%tennis%')
```

Matching against multiple possible values

How can you retrieve the details of facilities with ID 1 and 5? Try to do it without using the OR operator.

```
SELECT * FROM facilities WHERE facid IN (1, 5);
```

```
query = Facility.select().where(Facility.facid.in_([1, 5]))

# OR:
query = Facility.select().where((Facility.facid == 1) |
                                (Facility.facid == 5))
```

Classify results into buckets

How can you produce a list of facilities, with each labelled as 'cheap' or 'expensive' depending on if their monthly maintenance cost is more than $100? Return the name and monthly maintenance of the facilities in question.

```
SELECT name,
CASE WHEN monthlymaintenance > 100 THEN 'expensive' ELSE 'cheap' END
FROM facilities;
```

```
cost = Case(None, [(Facility.monthlymaintenance > 100, 'expensive')], 'cheap')
query = Facility.select(Facility.name, cost.alias('cost'))
```

Note: See documentation *Case* for more examples.

Working with dates

How can you produce a list of members who joined after the start of September 2012? Return the memid, surname, firstname, and joindate of the members in question.

```
SELECT memid, surname, firstname, joindate FROM members
WHERE joindate >= '2012-09-01';
```

```
query = (Member
         .select(Member.memid, Member.surname, Member.firstname, Member.joindate)
         .where(Member.joindate >= datetime.date(2012, 9, 1)))
```

Removing duplicates, and ordering results

How can you produce an ordered list of the first 10 surnames in the members table? The list must not contain duplicates.

```
SELECT DISTINCT surname FROM members ORDER BY surname LIMIT 10;
```

```
query = (Member
         .select(Member.surname)
         .order_by(Member.surname)
         .limit(10)
         .distinct())
```

Combining results from multiple queries

You, for some reason, want a combined list of all surnames and all facility names.

```
SELECT surname FROM members UNION SELECT name FROM facilities;
```

```
lhs = Member.select(Member.surname)
rhs = Facility.select(Facility.name)
query = lhs | rhs
```

Queries can be composed using the following operators:

- `|` - UNION
- `+` - UNION ALL
- `&` - INTERSECT
- `-` - EXCEPT

Simple aggregation

You'd like to get the signup date of your last member. How can you retrieve this information?

```
SELECT MAX(join_date) FROM members;
```

```
query = Member.select(fn.MAX(Member.joindate))
# To conveniently obtain a single scalar value, use "scalar()":
# max_join_date = query.scalar()
```

More aggregation

You'd like to get the first and last name of the last member(s) who signed up - not just the date.

```
SELECT firstname, surname, joindate FROM members
WHERE joindate = (SELECT MAX(joindate) FROM members);
```

```
# Use "alias()" to reference the same table multiple times in a query.
MemberAlias = Member.alias()
subq = MemberAlias.select(fn.MAX(MemberAlias.joindate))
query = (Member
         .select(Member.firstname, Member.surname, Member.joindate)
         .where(Member.joindate == subq))
```

1.14.4 Joins and Subqueries

This category deals primarily with a foundational concept in relational database systems: joining. Joining allows you to combine related information from multiple tables to answer a question. This isn't just beneficial for ease of querying: a lack of join capability encourages denormalisation of data, which increases the complexity of keeping your data internally consistent.

This topic covers inner, outer, and self joins, as well as spending a little time on subqueries (queries within queries).

Retrieve the start times of members' bookings

How can you produce a list of the start times for bookings by members named 'David Farrell'?

```
SELECT starttime FROM bookings
INNER JOIN members ON (bookings.memid = members.memid)
WHERE surname = 'Farrell' AND firstname = 'David';
```

```
query = (Booking
         .select(Booking.starttime)
         .join(Member)
         .where((Member.surname == 'Farrell') &
                (Member.firstname == 'David')))
```

Work out the start times of bookings for tennis courts

How can you produce a list of the start times for bookings for tennis courts, for the date '2012-09-21'? Return a list of start time and facility name pairings, ordered by the time.

```
SELECT starttime, name
FROM bookings
INNER JOIN facilities ON (bookings.facid = facilities.facid)
WHERE date_trunc('day', starttime) = '2012-09-21':: date
  AND name ILIKE 'tennis%'
ORDER BY starttime, name;
```

```
query = (Booking
         .select(Booking.starttime, Facility.name)
         .join(Facility)
         .where(
             (fn.date_trunc('day', Booking.starttime) == datetime.date(2012, 9, 21)) &
             Facility.name.startswith('Tennis'))
         .order_by(Booking.starttime, Facility.name))

# To retrieve the joined facility's name when iterating:
for booking in query:
    print(booking.starttime, booking.facility.name)
```

Produce a list of all members who have recommended another member

How can you output a list of all members who have recommended another member? Ensure that there are no duplicates in the list, and that results are ordered by (surname, firstname).

```
SELECT DISTINCT m.firstname, m.surname
FROM members AS m2
INNER JOIN members AS m ON (m.memid = m2.recommendedby)
ORDER BY m.surname, m.firstname;
```

```
MA = Member.alias()
query = (Member
         .select(Member.firstname, Member.surname)
         .join(MA, on=(MA.recommendedby == Member.memid))
         .order_by(Member.surname, Member.firstname))
```

Produce a list of all members, along with their recommender

How can you output a list of all members, including the individual who recommended them (if any)? Ensure that results are ordered by (surname, firstname).

```
SELECT m.firstname, m.surname, r.firstname, r.surname
FROM members AS m
LEFT OUTER JOIN members AS r ON (m.recommendedby = r.memid)
ORDER BY m.surname, m.firstname
```

```
MA = Member.alias()
query = (Member
         .select(Member.firstname, Member.surname, MA.firstname, MA.surname)
         .join(MA, JOIN.LEFT_OUTER, on=(Member.recommendedby == MA.memid))
         .order_by(Member.surname, Member.firstname))

# To display the recommender's name when iterating:
for m in query:
    print(m.firstname, m.surname)
    if m.recommendedby:
        print('  ', m.recommendedby.firstname, m.recommendedby.surname)
```

Produce a list of all members who have used a tennis court

How can you produce a list of all members who have used a tennis court? Include in your output the name of the court, and the name of the member formatted as a single column. Ensure no duplicate data, and order by the member name.

```
SELECT DISTINCT m.firstname || ' ' || m.surname AS member, f.name AS facility
FROM members AS m
INNER JOIN bookings AS b ON (m.memid = b.memid)
INNER JOIN facilities AS f ON (b.facid = f.facid)
WHERE f.name LIKE 'Tennis%'
ORDER BY member, facility;
```

```
fullname = Member.firstname + ' ' + Member.surname
query = (Member
         .select(fullname.alias('member'), Facility.name.alias('facility'))
         .join(Booking)
         .join(Facility)
         .where(Facility.name.startswith('Tennis'))
         .order_by(fullname, Facility.name)
         .distinct())
```

Produce a list of costly bookings

How can you produce a list of bookings on the day of 2012-09-14 which will cost the member (or guest) more than $30? Remember that guests have different costs to members (the listed costs are per half-hour 'slot'), and the guest user is always ID 0. Include in your output the name of the facility, the name of the member formatted as a single column, and the cost. Order by descending cost, and do not use any subqueries.

```
SELECT m.firstname || ' ' || m.surname AS member,
       f.name AS facility,
       (CASE WHEN m.memid = 0 THEN f.guestcost * b.slots
```

(continues on next page)

(continued from previous page)

```
        ELSE f.membercost * b.slots END) AS cost
FROM members AS m
INNER JOIN bookings AS b ON (m.memid = b.memid)
INNER JOIN facilities AS f ON (b.facid = f.facid)
WHERE (date_trunc('day', b.starttime) = '2012-09-14') AND
 ((m.memid = 0 AND b.slots * f.guestcost > 30) OR
  (m.memid > 0 AND b.slots * f.membercost > 30))
ORDER BY cost DESC;
```

```
cost = Case(Member.memid, (
    (0, Booking.slots * Facility.guestcost),
), (Booking.slots * Facility.membercost))
fullname = Member.firstname + ' ' + Member.surname

query = (Member
         .select(fullname.alias('member'), Facility.name.alias('facility'),
                 cost.alias('cost'))
         .join(Booking)
         .join(Facility)
         .where(
             (fn.date_trunc('day', Booking.starttime) == datetime.date(2012, 9, 14)) &
             (cost > 30))
         .order_by(SQL('cost').desc()))

# To iterate over the results, it might be easiest to use namedtuples:
for row in query.namedtuples():
    print(row.member, row.facility, row.cost)
```

Produce a list of all members, along with their recommender, using no joins.

How can you output a list of all members, including the individual who recommended them (if any), without using any joins? Ensure that there are no duplicates in the list, and that each firstname + surname pairing is formatted as a column and ordered.

```
SELECT DISTINCT m.firstname || ' ' || m.surname AS member,
   (SELECT r.firstname || ' ' || r.surname
    FROM cd.members AS r
    WHERE m.recommendedby = r.memid) AS recommended
FROM members AS m ORDER BY member;
```

```
MA = Member.alias()
subq = (MA
        .select(MA.firstname + ' ' + MA.surname)
        .where(Member.recommendedby == MA.memid))
query = (Member
         .select(fullname.alias('member'), subq.alias('recommended'))
         .order_by(fullname))
```

Produce a list of costly bookings, using a subquery

The "Produce a list of costly bookings" exercise contained some messy logic: we had to calculate the booking cost in both the WHERE clause and the CASE statement. Try to simplify this calculation using subqueries.

```
SELECT member, facility, cost from (
  SELECT
  m.firstname || ' ' || m.surname as member,
  f.name as facility,
  CASE WHEN m.memid = 0 THEN b.slots * f.guestcost
  ELSE b.slots * f.membercost END AS cost
  FROM members AS m
  INNER JOIN bookings AS b ON m.memid = b.memid
  INNER JOIN facilities AS f ON b.facid = f.facid
  WHERE date_trunc('day', b.starttime) = '2012-09-14'
) as bookings
WHERE cost > 30
ORDER BY cost DESC;
```

```
cost = Case(Member.memid, (
    (0, Booking.slots * Facility.guestcost),
), (Booking.slots * Facility.membercost))

iq = (Member
      .select(fullname.alias('member'), Facility.name.alias('facility'),
              cost.alias('cost'))
      .join(Booking)
      .join(Facility)
      .where(fn.date_trunc('day', Booking.starttime) == datetime.date(2012, 9, 14)))

query = (Member
         .select(iq.c.member, iq.c.facility, iq.c.cost)
         .from_(iq)
         .where(iq.c.cost > 30)
         .order_by(SQL('cost').desc()))

# To iterate, try using dicts:
for row in query.dicts():
    print(row['member'], row['facility'], row['cost'])
```

1.14.5 Modifying Data

Querying data is all well and good, but at some point you're probably going to want to put data into your database! This section deals with inserting, updating, and deleting information. Operations that alter your data like this are collectively known as Data Manipulation Language, or DML.

In previous sections, we returned to you the results of the query you've performed. Since modifications like the ones we're making in this section don't return any query results, we instead show you the updated content of the table you're supposed to be working on.

Insert some data into a table

The club is adding a new facility - a spa. We need to add it into the facilities table. Use the following values: facid: 9, Name: 'Spa', membercost: 20, guestcost: 30, initialoutlay: 100000, monthlymaintenance: 800

```
INSERT INTO "facilities" ("facid", "name", "membercost", "guestcost",
"initialoutlay", "monthlymaintenance") VALUES (9, 'Spa', 20, 30, 100000, 800)
```

```
res = Facility.insert({
    Facility.facid: 9,
    Facility.name: 'Spa',
    Facility.membercost: 20,
    Facility.guestcost: 30,
    Facility.initialoutlay: 100000,
    Facility.monthlymaintenance: 800}).execute()

# OR:
res = (Facility
       .insert(facid=9, name='Spa', membercost=20, guestcost=30,
               initialoutlay=100000, monthlymaintenance=800)
       .execute())
```

Insert multiple rows of data into a table

In the previous exercise, you learned how to add a facility. Now you're going to add multiple facilities in one command. Use the following values:

facid: 9, Name: 'Spa', membercost: 20, guestcost: 30, initialoutlay: 100000, monthlymaintenance: 800.

facid: 10, Name: 'Squash Court 2', membercost: 3.5, guestcost: 17.5, initialoutlay: 5000, monthlymaintenance: 80.

```
-- see above --
```

```
data = [
    {'facid': 9, 'name': 'Spa', 'membercost': 20, 'guestcost': 30,
     'initialoutlay': 100000, 'monthlymaintenance': 800},
    {'facid': 10, 'name': 'Squash Court 2', 'membercost': 3.5,
     'guestcost': 17.5, 'initialoutlay': 5000, 'monthlymaintenance': 80}]
res = Facility.insert_many(data).execute()
```

Insert calculated data into a table

Let's try adding the spa to the facilities table again. This time, though, we want to automatically generate the value for the next facid, rather than specifying it as a constant. Use the following values for everything else: Name: 'Spa', membercost: 20, guestcost: 30, initialoutlay: 100000, monthlymaintenance: 800.

```
INSERT INTO "facilities" ("facid", "name", "membercost", "guestcost",
  "initialoutlay", "monthlymaintenance")
SELECT (SELECT (MAX("facid") + 1) FROM "facilities") AS _,
        'Spa', 20, 30, 100000, 800;
```

```
maxq = Facility.select(fn.MAX(Facility.facid) + 1)
subq = Select(columns=(maxq, 'Spa', 20, 30, 100000, 800))
res = Facility.insert_from(subq, Facility._meta.sorted_fields).execute()
```

Update some existing data

We made a mistake when entering the data for the second tennis court. The initial outlay was 10000 rather than 8000: you need to alter the data to fix the error.

```
UPDATE facilities SET initialoutlay = 10000 WHERE name = 'Tennis Court 2';
```

```
res = (Facility
       .update({Facility.initialoutlay: 10000})
       .where(Facility.name == 'Tennis Court 2')
       .execute())

# OR:
res = (Facility
       .update(initialoutlay=10000)
       .where(Facility.name == 'Tennis Court 2')
       .execute())
```

Update multiple rows and columns at the same time

We want to increase the price of the tennis courts for both members and guests. Update the costs to be 6 for members, and 30 for guests.

```
UPDATE facilities SET membercost=6, guestcost=30 WHERE name ILIKE 'Tennis%';
```

```
nrows = (Facility
         .update(membercost=6, guestcost=30)
         .where(Facility.name.startswith('Tennis'))
         .execute())
```

Update a row based on the contents of another row

We want to alter the price of the second tennis court so that it costs 10% more than the first one. Try to do this without using constant values for the prices, so that we can reuse the statement if we want to.

```
UPDATE facilities SET
membercost = (SELECT membercost * 1.1 FROM facilities WHERE facid = 0),
guestcost = (SELECT guestcost * 1.1 FROM facilities WHERE facid = 0)
WHERE facid = 1;

-- OR --
WITH new_prices (nmc, ngc) AS (
  SELECT membercost * 1.1, guestcost * 1.1
  FROM facilities WHERE name = 'Tennis Court 1')
UPDATE facilities
SET membercost = new_prices.nmc, guestcost = new_prices.ngc
FROM new_prices
WHERE name = 'Tennis Court 2'
```

```
sq1 = Facility.select(Facility.membercost * 1.1).where(Facility.facid == 0)
sq2 = Facility.select(Facility.guestcost * 1.1).where(Facility.facid == 0)

res = (Facility
       .update(membercost=sq1, guestcost=sq2)
       .where(Facility.facid == 1)
       .execute())

# OR:
```

(continues on next page)

(continued from previous page)

```
cte = (Facility
       .select(Facility.membercost * 1.1, Facility.guestcost * 1.1)
       .where(Facility.name == 'Tennis Court 1')
       .cte('new_prices', columns=('nmc', 'ngc')))
res = (Facility
       .update(membercost=SQL('new_prices.nmc'), guestcost=SQL('new_prices.ngc'))
       .with_cte(cte)
       .from_(cte)
       .where(Facility.name == 'Tennis Court 2')
       .execute())
```

Delete all bookings

As part of a clearout of our database, we want to delete all bookings from the bookings table.

```
DELETE FROM bookings;
```

```
nrows = Booking.delete().execute()
```

Delete a member from the cd.members table

We want to remove member 37, who has never made a booking, from our database.

```
DELETE FROM members WHERE memid = 37;
```

```
nrows = Member.delete().where(Member.memid == 37).execute()
```

Delete based on a subquery

How can we make that more general, to delete all members who have never made a booking?

```
DELETE FROM members WHERE NOT EXISTS (
  SELECT * FROM bookings WHERE bookings.memid = members.memid);
```

```
subq = Booking.select().where(Booking.member == Member.memid)
nrows = Member.delete().where(~fn.EXISTS(subq)).execute()
```

1.14.6 Aggregation

Aggregation is one of those capabilities that really make you appreciate the power of relational database systems. It allows you to move beyond merely persisting your data, into the realm of asking truly interesting questions that can be used to inform decision making. This category covers aggregation at length, making use of standard grouping as well as more recent window functions.

Count the number of facilities

For our first foray into aggregates, we're going to stick to something simple. We want to know how many facilities exist - simply produce a total count.

```
SELECT COUNT(facid) FROM facilities;
```

```
query = Facility.select(fn.COUNT(Facility.facid))
count = query.scalar()

# OR:
count = Facility.select().count()
```

Count the number of expensive facilities

Produce a count of the number of facilities that have a cost to guests of 10 or more.

```
SELECT COUNT(facid) FROM facilities WHERE guestcost >= 10
```

```
query = Facility.select(fn.COUNT(Facility.facid)).where(Facility.guestcost >= 10)
count = query.scalar()

# OR:
# count = Facility.select().where(Facility.guestcost >= 10).count()
```

Count the number of recommendations each member makes.

Produce a count of the number of recommendations each member has made. Order by member ID.

```
SELECT recommendedby, COUNT(memid) FROM members
WHERE recommendedby IS NOT NULL
GROUP BY recommendedby
ORDER BY recommendedby
```

```
query = (Member
         .select(Member.recommendedby, fn.COUNT(Member.memid))
         .where(Member.recommendedby.is_null(False))
         .group_by(Member.recommendedby)
         .order_by(Member.recommendedby))
```

List the total slots booked per facility

Produce a list of the total number of slots booked per facility. For now, just produce an output table consisting of facility id and slots, sorted by facility id.

```
SELECT facid, SUM(slots) FROM bookings GROUP BY facid ORDER BY facid;
```

```
query = (Booking
         .select(Booking.facid, fn.SUM(Booking.slots))
         .group_by(Booking.facid)
         .order_by(Booking.facid))
```

List the total slots booked per facility in a given month

Produce a list of the total number of slots booked per facility in the month of September 2012. Produce an output table consisting of facility id and slots, sorted by the number of slots.

```
SELECT facid, SUM(slots)
FROM bookings
WHERE (date_trunc('month', starttime) = '2012-09-01'::dates)
GROUP BY facid
ORDER BY SUM(slots)
```

```
query = (Booking
         .select(Booking.facility, fn.SUM(Booking.slots))
         .where(fn.date_trunc('month', Booking.starttime) == datetime.date(2012, 9,␣
↪1))
         .group_by(Booking.facility)
         .order_by(fn.SUM(Booking.slots)))
```

List the total slots booked per facility per month

Produce a list of the total number of slots booked per facility per month in the year of 2012. Produce an output table consisting of facility id and slots, sorted by the id and month.

```
SELECT facid, date_part('month', starttime), SUM(slots)
FROM bookings
WHERE date_part('year', starttime) = 2012
GROUP BY facid, date_part('month', starttime)
ORDER BY facid, date_part('month', starttime)
```

```
month = fn.date_part('month', Booking.starttime)
query = (Booking
         .select(Booking.facility, month, fn.SUM(Booking.slots))
         .where(fn.date_part('year', Booking.starttime) == 2012)
         .group_by(Booking.facility, month)
         .order_by(Booking.facility, month))
```

Find the count of members who have made at least one booking

Find the total number of members who have made at least one booking.

```
SELECT COUNT(DISTINCT memid) FROM bookings

-- OR --
SELECT COUNT(1) FROM (SELECT DISTINCT memid FROM bookings) AS _
```

```
query = Booking.select(fn.COUNT(Booking.member.distinct()))

# OR:
query = Booking.select(Booking.member).distinct()
count = query.count()  # count() wraps in SELECT COUNT(1) FROM (...)
```

List facilities with more than 1000 slots booked

Produce a list of facilities with more than 1000 slots booked. Produce an output table consisting of facility id and hours, sorted by facility id.

```
SELECT facid, SUM(slots) FROM bookings
GROUP BY facid
HAVING SUM(slots) > 1000
ORDER BY facid;
```

```
query = (Booking
         .select(Booking.facility, fn.SUM(Booking.slots))
         .group_by(Booking.facility)
         .having(fn.SUM(Booking.slots) > 1000)
         .order_by(Booking.facility))
```

Find the total revenue of each facility

Produce a list of facilities along with their total revenue. The output table should consist of facility name and revenue, sorted by revenue. Remember that there's a different cost for guests and members!

```
SELECT f.name, SUM(b.slots * (
CASE WHEN b.memid = 0 THEN f.guestcost ELSE f.membercost END)) AS revenue
FROM bookings AS b
INNER JOIN facilities AS f ON b.facid = f.facid
GROUP BY f.name
ORDER BY revenue;
```

```
revenue = fn.SUM(Booking.slots * Case(None, (
    (Booking.member == 0, Facility.guestcost),
), Facility.membercost))

query = (Facility
         .select(Facility.name, revenue.alias('revenue'))
         .join(Booking)
         .group_by(Facility.name)
         .order_by(SQL('revenue')))
```

Find facilities with a total revenue less than 1000

Produce a list of facilities with a total revenue less than 1000. Produce an output table consisting of facility name and revenue, sorted by revenue. Remember that there's a different cost for guests and members!

```
SELECT f.name, SUM(b.slots * (
CASE WHEN b.memid = 0 THEN f.guestcost ELSE f.membercost END)) AS revenue
FROM bookings AS b
INNER JOIN facilities AS f ON b.facid = f.facid
GROUP BY f.name
HAVING SUM(b.slots * ...) < 1000
ORDER BY revenue;
```

```
# Same definition as previous example.
revenue = fn.SUM(Booking.slots * Case(None, (
    (Booking.member == 0, Facility.guestcost),
), Facility.membercost))

query = (Facility
         .select(Facility.name, revenue.alias('revenue'))
```

(continues on next page)

(continued from previous page)

```
         .join(Booking)
         .group_by(Facility.name)
         .having(revenue < 1000)
         .order_by(SQL('revenue')))
```

Output the facility id that has the highest number of slots booked

Output the facility id that has the highest number of slots booked.

```
SELECT facid, SUM(slots) FROM bookings
GROUP BY facid
ORDER BY SUM(slots) DESC
LIMIT 1
```

```
query = (Booking
         .select(Booking.facility, fn.SUM(Booking.slots))
         .group_by(Booking.facility)
         .order_by(fn.SUM(Booking.slots).desc())
         .limit(1))

# Retrieve multiple scalar values by calling scalar() with as_tuple=True.
facid, nslots = query.scalar(as_tuple=True)
```

List the total slots booked per facility per month, part 2

Produce a list of the total number of slots booked per facility per month in the year of 2012. In this version, include output rows containing totals for all months per facility, and a total for all months for all facilities. The output table should consist of facility id, month and slots, sorted by the id and month. When calculating the aggregated values for all months and all facids, return null values in the month and facid columns.

Postgres ONLY.

```
SELECT facid, date_part('month', starttime), SUM(slots)
FROM booking
WHERE date_part('year', starttime) = 2012
GROUP BY ROLLUP(facid, date_part('month', starttime))
ORDER BY facid, date_part('month', starttime)
```

```
month = fn.date_part('month', Booking.starttime)
query = (Booking
         .select(Booking.facility,
                 month.alias('month'),
                 fn.SUM(Booking.slots))
         .where(fn.date_part('year', Booking.starttime) == 2012)
         .group_by(fn.ROLLUP(Booking.facility, month))
         .order_by(Booking.facility, month))
```

List the total hours booked per named facility

Produce a list of the total number of hours booked per facility, remembering that a slot lasts half an hour. The output table should consist of the facility id, name, and hours booked, sorted by facility id.

```
SELECT f.facid, f.name, SUM(b.slots) * .5
FROM facilities AS f
INNER JOIN bookings AS b ON (f.facid = b.facid)
GROUP BY f.facid, f.name
ORDER BY f.facid
```

```
query = (Facility
         .select(Facility.facid, Facility.name, fn.SUM(Booking.slots) * .5)
         .join(Booking)
         .group_by(Facility.facid, Facility.name)
         .order_by(Facility.facid))
```

List each member's first booking after September 1st 2012

Produce a list of each member name, id, and their first booking after September 1st 2012. Order by member ID.

```
SELECT m.surname, m.firstname, m.memid, min(b.starttime) as starttime
FROM members AS m
INNER JOIN bookings AS b ON b.memid = m.memid
WHERE starttime >= '2012-09-01'
GROUP BY m.surname, m.firstname, m.memid
ORDER BY m.memid;
```

```
query = (Member
         .select(Member.surname, Member.firstname, Member.memid,
                 fn.MIN(Booking.starttime).alias('starttime'))
         .join(Booking)
         .where(Booking.starttime >= datetime.date(2012, 9, 1))
         .group_by(Member.surname, Member.firstname, Member.memid)
         .order_by(Member.memid))
```

Produce a list of member names, with each row containing the total member count

Produce a list of member names, with each row containing the total member count. Order by join date.

Postgres ONLY (as written).

```
SELECT COUNT(*) OVER(), firstname, surname
FROM members ORDER BY joindate
```

```
query = (Member
         .select(fn.COUNT(Member.memid).over(), Member.firstname,
                 Member.surname)
         .order_by(Member.joindate))
```

Produce a numbered list of members

Produce a monotonically increasing numbered list of members, ordered by their date of joining. Remember that member IDs are not guaranteed to be sequential.

Postgres ONLY (as written).

```
SELECT row_number() OVER (ORDER BY joindate), firstname, surname
FROM members ORDER BY joindate;
```

```
query = (Member
         .select(fn.row_number().over(order_by=[Member.joindate]),
                 Member.firstname, Member.surname)
         .order_by(Member.joindate))
```

Output the facility id that has the highest number of slots booked, again

Output the facility id that has the highest number of slots booked. Ensure that in the event of a tie, all tieing results get output.

Postgres ONLY (as written).

```
SELECT facid, total FROM (
  SELECT facid, SUM(slots) AS total,
         rank() OVER (order by SUM(slots) DESC) AS rank
  FROM bookings
  GROUP BY facid
) AS ranked WHERE rank = 1
```

```
rank = fn.rank().over(order_by=[fn.SUM(Booking.slots).desc()])

subq = (Booking
        .select(Booking.facility, fn.SUM(Booking.slots).alias('total'),
                rank.alias('rank'))
        .group_by(Booking.facility))

# Here we use a plain Select() to create our query.
query = (Select(columns=[subq.c.facid, subq.c.total])
         .from_(subq)
         .where(subq.c.rank == 1)
         .bind(db))  # We must bind() it to the database.

# To iterate over the query results:
for facid, total in query.tuples():
    print(facid, total)
```

Rank members by (rounded) hours used

Produce a list of members, along with the number of hours they've booked in facilities, rounded to the nearest ten hours. Rank them by this rounded figure, producing output of first name, surname, rounded hours, rank. Sort by rank, surname, and first name.

Postgres ONLY (as written).

```
SELECT firstname, surname,
((SUM(bks.slots)+10)/20)*10 as hours,
rank() over (order by ((sum(bks.slots)+10)/20)*10 desc) as rank
FROM members AS mems
INNER JOIN bookings AS bks ON mems.memid = bks.memid
GROUP BY mems.memid
ORDER BY rank, surname, firstname;
```

```
hours = ((fn.SUM(Booking.slots) + 10) / 20) * 10
query = (Member
         .select(Member.firstname, Member.surname, hours.alias('hours'),
                 fn.rank().over(order_by=[hours.desc()]).alias('rank'))
         .join(Booking)
         .group_by(Member.memid)
         .order_by(SQL('rank'), Member.surname, Member.firstname))
```

Find the top three revenue generating facilities

Produce a list of the top three revenue generating facilities (including ties). Output facility name and rank, sorted by rank and facility name.

Postgres ONLY (as written).

```
SELECT name, rank FROM (
    SELECT f.name, RANK() OVER (ORDER BY SUM(
        CASE WHEN memid = 0 THEN slots * f.guestcost
        ELSE slots * f.membercost END) DESC) AS rank
    FROM bookings
    INNER JOIN facilities AS f ON bookings.facid = f.facid
    GROUP BY f.name) AS subq
WHERE rank <= 3
ORDER BY rank;
```

```
total_cost = fn.SUM(Case(None, (
    (Booking.member == 0, Booking.slots * Facility.guestcost),
), (Booking.slots * Facility.membercost)))

subq = (Facility
        .select(Facility.name,
                fn.RANK().over(order_by=[total_cost.desc()]).alias('rank'))
        .join(Booking)
        .group_by(Facility.name))

query = (Select(columns=[subq.c.name, subq.c.rank])
         .from_(subq)
         .where(subq.c.rank <= 3)
         .order_by(subq.c.rank)
         .bind(db))  # Here again we used plain Select, and call bind().
```

Classify facilities by value

Classify facilities into equally sized groups of high, average, and low based on their revenue. Order by classification and facility name.

Postgres ONLY (as written).

```
SELECT name,
  CASE class WHEN 1 THEN 'high' WHEN 2 THEN 'average' ELSE 'low' END
FROM (
  SELECT f.name, ntile(3) OVER (ORDER BY SUM(
    CASE WHEN memid = 0 THEN slots * f.guestcost ELSE slots * f.membercost
    END) DESC) AS class
```

(continues on next page)

(continued from previous page)

```
  FROM bookings INNER JOIN facilities AS f ON bookings.facid = f.facid
  GROUP BY f.name
) AS subq
ORDER BY class, name;
```

```
cost = fn.SUM(Case(None, (
    (Booking.member == 0, Booking.slots * Facility.guestcost),
), (Booking.slots * Facility.membercost)))
subq = (Facility
        .select(Facility.name,
                fn.NTILE(3).over(order_by=[cost.desc()]).alias('klass'))
        .join(Booking)
        .group_by(Facility.name))

klass_case = Case(subq.c.klass, [(1, 'high'), (2, 'average')], 'low')
query = (Select(columns=[subq.c.name, klass_case])
         .from_(subq)
         .order_by(subq.c.klass, subq.c.name)
         .bind(db))
```

1.14.7 Recursion

Common Table Expressions allow us to, effectively, create our own temporary tables for the duration of a query - they're largely a convenience to help us make more readable SQL. Using the WITH RECURSIVE modifier, however, it's possible for us to create recursive queries. This is enormously advantageous for working with tree and graph-structured data - imagine retrieving all of the relations of a graph node to a given depth, for example.

Find the upward recommendation chain for member ID 27

Find the upward recommendation chain for member ID 27: that is, the member who recommended them, and the member who recommended that member, and so on. Return member ID, first name, and surname. Order by descending member id.

```
WITH RECURSIVE recommenders(recommender) as (
  SELECT recommendedby FROM members WHERE memid = 27
  UNION ALL
  SELECT mems.recommendedby
  FROM recommenders recs
  INNER JOIN members AS mems ON mems.memid = recs.recommender
)
SELECT recs.recommender, mems.firstname, mems.surname
FROM recommenders AS recs
INNER JOIN members AS mems ON recs.recommender = mems.memid
ORDER By memid DESC;
```

```
# Base-case of recursive CTE. Get member recommender where memid=27.
base = (Member
        .select(Member.recommendedby)
        .where(Member.memid == 27)
        .cte('recommenders', recursive=True, columns=('recommender',)))

# Recursive term of CTE. Get recommender of previous recommender.
```

(continues on next page)

(continued from previous page)

```
MA = Member.alias()
recursive = (MA
             .select(MA.recommendedby)
             .join(base, on=(MA.memid == base.c.recommender)))

# Combine the base-case with the recursive term.
cte = base.union_all(recursive)

# Select from the recursive CTE, joining on member to get name info.
query = (cte
         .select_from(cte.c.recommender, Member.firstname, Member.surname)
         .join(Member, on=(cte.c.recommender == Member.memid))
         .order_by(Member.memid.desc()))
```

1.15 Query Builder

Peewee's high-level *Model* and *Field* APIs are built upon lower-level *Table* and *Column* counterparts. While these lower-level APIs are not documented in as much detail as their high-level counterparts, this document will present an overview with examples that should hopefully allow you to experiment.

We'll use the following schema:

```
CREATE TABLE "person" (
    "id" INTEGER NOT NULL PRIMARY KEY,
    "first" TEXT NOT NULL,
    "last" TEXT NOT NULL);

CREATE TABLE "note" (
    "id" INTEGER NOT NULL PRIMARY KEY,
    "person_id" INTEGER NOT NULL,
    "content" TEXT NOT NULL,
    "timestamp" DATETIME NOT NULL,
    FOREIGN KEY ("person_id") REFERENCES "person" ("id"));

CREATE TABLE "reminder" (
    "id" INTEGER NOT NULL PRIMARY KEY,
    "note_id" INTEGER NOT NULL,
    "alarm" DATETIME NOT NULL,
    FOREIGN KEY ("note_id") REFERENCES "note" ("id"));
```

1.15.1 Declaring tables

There are two ways we can declare *Table* objects for working with these tables:

```
# Explicitly declare columns
Person = Table('person', ('id', 'first', 'last'))

Note = Table('note', ('id', 'person_id', 'content', 'timestamp'))

# Do not declare columns, they will be accessed using magic ".c" attribute
Reminder = Table('reminder')
```

Typically we will want to *bind()* our tables to a database. This saves us having to pass the database explicitly every time we wish to execute a query on the table:

```
db = SqliteDatabase('my_app.db')
Person = Person.bind(db)
Note = Note.bind(db)
Reminder = Reminder.bind(db)
```

1.15.2 Select queries

To select the first three notes and print their content, we can write:

```
query = Note.select().order_by(Note.timestamp).limit(3)
for note_dict in query:
    print(note_dict['content'])
```

Note: By default, rows will be returned as dictionaries. You can use the *tuples()*, *namedtuples()* or *objects()* methods to specify a different container for the row data, if you wish.

Because we didn't specify any columns, all the columns we defined in the note's *Table* constructor will be selected. This won't work for Reminder, as we didn't specify any columns at all.

To select all notes published in 2018 along with the name of the creator, we will use `join()`. We'll also request that rows be returned as *namedtuple* objects:

```
query = (Note
         .select(Note.content, Note.timestamp, Person.first, Person.last)
         .join(Person, on=(Note.person_id == Person.id))
         .where(Note.timestamp >= datetime.date(2018, 1, 1))
         .order_by(Note.timestamp)
         .namedtuples())

for row in query:
    print(row.timestamp, '-', row.content, '-', row.first, row.last)
```

Let's query for the most prolific people, that is, get the people who have created the most notes. This introduces calling a SQL function (COUNT), which is accomplished using the `fn` object:

```
name = Person.first.concat(' ').concat(Person.last)
query = (Person
         .select(name.alias('name'), fn.COUNT(Note.id).alias('count'))
         .join(Note, JOIN.LEFT_OUTER, on=(Note.person_id == Person.id))
         .group_by(name)
         .order_by(fn.COUNT(Note.id).desc()))
for row in query:
    print(row['name'], row['count'])
```

There are a couple things to note in the above query:

- We store an expression in a variable (`name`), then use it in the query.
- We call SQL functions using `fn.<function>(...)` passing arguments as if it were a normal Python function.
- The *alias()* method is used to specify the name used for a column or calculation.

As a more complex example, we'll generate a list of all people and the contents and timestamp of their most recently-published note. To do this, we will end up using the Note table twice in different contexts within the same query, which will require us to use a table alias.

```
# Start with the query that calculates the timestamp of the most recent
# note for each person.
NA = Note.alias('na')
max_note = (NA
            .select(NA.person_id, fn.MAX(NA.timestamp).alias('max_ts'))
            .group_by(NA.person_id)
            .alias('max_note'))

# Now we'll select from the note table, joining on both the subquery and
# on the person table to construct the result set.
query = (Note
         .select(Note.content, Note.timestamp, Person.first, Person.last)
         .join(max_note, on=((max_note.c.person_id == Note.person_id) &
                             (max_note.c.max_ts == Note.timestamp)))
         .join(Person, on=(Note.person_id == Person.id))
         .order_by(Person.first, Person.last))

for row in query.namedtuples():
    print(row.first, row.last, ':', row.timestamp, '-', row.content)
```

In the join predicate for the join on the *max_note* subquery, we can reference columns in the subquery using the magical ".c" attribute. So, *max_note.c.max_ts* is translated into "the max_ts column value from the max_note subquery".

We can also use the ".c" magic attribute to access columns on tables that do not explicitly define their columns, like we did with the Reminder table. Here's a simple query to get all reminders for today, along with their associated note content:

```
today = datetime.date.today()
tomorrow = today + datetime.timedelta(days=1)

query = (Reminder
         .select(Reminder.c.alarm, Note.content)
         .join(Note, on=(Reminder.c.note_id == Note.id))
         .where(Reminder.c.alarm.between(today, tomorrow))
         .order_by(Reminder.c.alarm))
for row in query:
    print(row['alarm'], row['content'])
```

Note: The ".c" attribute will not work on tables that explicitly define their columns, to prevent confusion.

1.15.3 Insert queries

Inserting data is straightforward. We can specify data to *insert()* in two different ways (in both cases, the ID of the new row is returned):

```
# Using keyword arguments:
zaizee_id = Person.insert(first='zaizee', last='cat').execute()

# Using column: value mappings:
Note.insert({
```

(continues on next page)

(continued from previous page)

```
    Note.person_id: zaizee_id,
    Note.content: 'meeeeowwww',
    Note.timestamp: datetime.datetime.now()}).execute()
```

It is easy to bulk-insert data, just pass in either:

- A list of dictionaries (all must have the same keys/columns).
- A list of tuples, if the columns are specified explicitly.

Examples:

```
people = [
    {'first': 'Bob', 'last': 'Foo'},
    {'first': 'Herb', 'last': 'Bar'},
    {'first': 'Nuggie', 'last': 'Bar'}]

# Inserting multiple rows returns the ID of the last-inserted row.
last_id = Person.insert(people).execute()

# We can also specify row tuples, so long as we tell Peewee which
# columns the tuple values correspond to:
people = [
    ('Bob', 'Foo'),
    ('Herb', 'Bar'),
    ('Nuggie', 'Bar')]
Person.insert(people, columns=[Person.first, Person.last]).execute()
```

1.15.4 Update queries

update() queries accept either keyword arguments or a dictionary mapping column to value, just like *insert()*.

Examples:

```
# "Bob" changed his last name from "Foo" to "Baze".
nrows = (Person
         .update(last='Baze')
         .where((Person.first == 'Bob') &
                (Person.last == 'Foo'))
         .execute())

# Use dictionary mapping column to value.
nrows = (Person
         .update({Person.last: 'Baze'})
         .where((Person.first == 'Bob') &
                (Person.last == 'Foo'))
         .execute())
```

You can also use expressions as the value to perform an atomic update. Imagine we have a *PageView* table and we need to atomically increment the page-view count for some URL:

```
# Do an atomic update:
(PageView
 .update({PageView.count: PageView.count + 1})
 .where(PageView.url == some_url)
 .execute())
```

1.15.5 Delete queries

delete() queries are simplest of all, as they do not accept any arguments:

```
# Delete all notes created before 2018, returning number deleted.
n = Note.delete().where(Note.timestamp < datetime.date(2018, 1, 1)).execute()
```

Because DELETE (and UPDATE) queries do not support joins, we can use subqueries to delete rows based on values in related tables. For example, here is how you would delete all notes by anyone whose last name is "Foo":

```
# Get the id of all people whose last name is "Foo".
foo_people = Person.select(Person.id).where(Person.last == 'Foo')

# Delete all notes by any person whose ID is in the previous query.
Note.delete().where(Note.person_id.in_(foo_people)).execute()
```

1.15.6 Query Objects

One of the fundamental limitations of the abstractions provided by Peewee 2.x was the absence of a class that represented a structured query with no relation to a given model class.

An example of this might be computing aggregate values over a subquery. For example, the *count()* method, which returns the count of rows in an arbitrary query, is implemented by wrapping the query:

```
SELECT COUNT(1) FROM (...)
```

To accomplish this with Peewee, the implementation is written in this way:

```
def count(query):
    # Select([source1, ... sourcen], [column1, ...columnn])
    wrapped = Select(from_list=[query], columns=[fn.COUNT(SQL('1'))])
    curs = wrapped.tuples().execute(db)
    return curs[0][0]  # Return first column from first row of result.
```

We can actually express this more concisely using the *scalar()* method, which is suitable for returning values from aggregate queries:

```
def count(query):
    wrapped = Select(from_list=[query], columns=[fn.COUNT(SQL('1'))])
    return wrapped.scalar(db)
```

The *Query Examples* document has a more complex example, in which we write a query for a facility with the highest number of available slots booked:

The SQL we wish to express is:

```
SELECT facid, total FROM (
  SELECT facid, SUM(slots) AS total,
         rank() OVER (order by SUM(slots) DESC) AS rank
  FROM bookings
  GROUP BY facid
) AS ranked
WHERE rank = 1
```

We can express this fairly elegantly by using a plain *Select* for the outer query:

```
# Store rank expression in variable for readability.
rank_expr = fn.rank().over(order_by=[fn.SUM(Booking.slots).desc()])

subq = (Booking
        .select(Booking.facility, fn.SUM(Booking.slots).alias('total'),
                rank_expr.alias('rank'))
        .group_by(Booking.facility))

# Use a plain "Select" to create outer query.
query = (Select(columns=[subq.c.facid, subq.c.total])
         .from_(subq)
         .where(subq.c.rank == 1)
         .tuples())

# Iterate over the resulting facility ID(s) and total(s):
for facid, total in query.execute(db):
    print(facid, total)
```

For another example, let's create a recursive common table expression to calculate the first 10 fibonacci numbers:

```
base = Select(columns=(
    Value(1).alias('n'),
    Value(0).alias('fib_n'),
    Value(1).alias('next_fib_n'))).cte('fibonacci', recursive=True)

n = (base.c.n + 1).alias('n')
recursive_term = Select(columns=(
    n,
    base.c.next_fib_n,
    base.c.fib_n + base.c.next_fib_n)).from_(base).where(n < 10)

fibonacci = base.union_all(recursive_term)
query = fibonacci.select_from(fibonacci.c.n, fibonacci.c.fib_n)

results = list(query.execute(db))

# Generates the following result list:
[{'fib_n': 0, 'n': 1},
 {'fib_n': 1, 'n': 2},
 {'fib_n': 1, 'n': 3},
 {'fib_n': 2, 'n': 4},
 {'fib_n': 3, 'n': 5},
 {'fib_n': 5, 'n': 6},
 {'fib_n': 8, 'n': 7},
 {'fib_n': 13, 'n': 8},
 {'fib_n': 21, 'n': 9},
 {'fib_n': 34, 'n': 10}]
```

1.15.7 More

For a description of the various classes used to describe a SQL AST, see the *query builder API documentation*.

If you're interested in learning more, you can also check out the project source code.

1.16 Hacks

Collected hacks using peewee. Have a cool hack you'd like to share? Open an issue on GitHub or contact me.

1.16.1 Optimistic Locking

Optimistic locking is useful in situations where you might ordinarily use a *SELECT FOR UPDATE* (or in SQLite, *BEGIN IMMEDIATE*). For example, you might fetch a user record from the database, make some modifications, then save the modified user record. Typically this scenario would require us to lock the user record for the duration of the transaction, from the moment we select it, to the moment we save our changes.

In optimistic locking, on the other hand, we do *not* acquire any lock and instead rely on an internal *version* column in the row we're modifying. At read time, we see what version the row is currently at, and on save, we ensure that the update takes place only if the version is the same as the one we initially read. If the version is higher, then some other process must have snuck in and changed the row – to save our modified version could result in the loss of important changes.

It's quite simple to implement optimistic locking in Peewee, here is a base class that you can use as a starting point:

```
from peewee import *

class ConflictDetectedException(Exception): pass

class BaseVersionedModel(Model):
    version = IntegerField(default=1, index=True)

    def save_optimistic(self):
        if not self.id:
            # This is a new record, so the default logic is to perform an
            # INSERT. Ideally your model would also have a unique
            # constraint that made it impossible for two INSERTs to happen
            # at the same time.
            return self.save()

        # Update any data that has changed and bump the version counter.
        field_data = dict(self.__data__)
        current_version = field_data.pop('version', 1)
        self._populate_unsaved_relations(field_data)
        field_data = self._prune_fields(field_data, self.dirty_fields)
        if not field_data:
            raise ValueError('No changes have been made.')

        ModelClass = type(self)
        field_data['version'] = ModelClass.version + 1  # Atomic increment.

        query = ModelClass.update(**field_data).where(
            (ModelClass.version == current_version) &
            (ModelClass.id == self.id))
        if query.execute() == 0:
            # No rows were updated, indicating another process has saved
            # a new version. How you handle this situation is up to you,
            # but for simplicity I'm just raising an exception.
            raise ConflictDetectedException()
        else:
            # Increment local version to match what is now in the db.
```

(continues on next page)

(continued from previous page)

```
            self.version += 1
            return True
```

Here's an example of how this works. Let's assume we have the following model definition. Note that there's a unique constraint on the username – this is important as it provides a way to prevent double-inserts.

```
class User(BaseVersionedModel):
    username = CharField(unique=True)
    favorite_animal = CharField()
```

Example:

```
>>> u = User(username='charlie', favorite_animal='cat')
>>> u.save_optimistic()
True

>>> u.version
1

>>> u.save_optimistic()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "x.py", line 18, in save_optimistic
    raise ValueError('No changes have been made.')
ValueError: No changes have been made.

>>> u.favorite_animal = 'kitten'
>>> u.save_optimistic()
True

# Simulate a separate thread coming in and updating the model.
>>> u2 = User.get(User.username == 'charlie')
>>> u2.favorite_animal = 'macaw'
>>> u2.save_optimistic()
True

# Now, attempt to change and re-save the original instance:
>>> u.favorite_animal = 'little parrot'
>>> u.save_optimistic()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "x.py", line 30, in save_optimistic
    raise ConflictDetectedException()
ConflictDetectedException: current version is out of sync
```

1.16.2 Top object per group

These examples describe several ways to query the single top item per group. For a thorough discuss of various techniques, check out my blog post Querying the top item by group with Peewee ORM. If you are interested in the more general problem of querying the top *N* items, see the section below *Top N objects per group*.

In these examples we will use the *User* and *Tweet* models to find each user and their most-recent tweet.

The most efficient method I found in my testing uses the `MAX()` aggregate function.

We will perform the aggregation in a non-correlated subquery, so we can be confident this method will be performant.

The idea is that we will select the posts, grouped by their author, whose timestamp is equal to the max observed timestamp for that user.

```
# When referencing a table multiple times, we'll call Model.alias() to create
# a secondary reference to the table.
TweetAlias = Tweet.alias()

# Create a subquery that will calculate the maximum Tweet created_date for each
# user.
subquery = (TweetAlias
            .select(
                TweetAlias.user,
                fn.MAX(TweetAlias.created_date).alias('max_ts'))
            .group_by(TweetAlias.user)
            .alias('tweet_max_subquery'))

# Query for tweets and join using the subquery to match the tweet's user
# and created_date.
query = (Tweet
         .select(Tweet, User)
         .join(User)
         .switch(Tweet)
         .join(subquery, on=(
             (Tweet.created_date == subquery.c.max_ts) &
             (Tweet.user == subquery.c.user_id))))
```

SQLite and MySQL are a bit more lax and permit grouping by a subset of the columns that are selected. This means we can do away with the subquery and express it quite concisely:

```
query = (Tweet
         .select(Tweet, User)
         .join(User)
         .group_by(Tweet.user)
         .having(Tweet.created_date == fn.MAX(Tweet.created_date)))
```

1.16.3 Top N objects per group

These examples describe several ways to query the top *N* items per group reasonably efficiently. For a thorough discussion of various techniques, check out my blog post Querying the top N objects per group with Peewee ORM.

In these examples we will use the *User* and *Tweet* models to find each user and their three most-recent tweets.

Postgres lateral joins

Lateral joins are a neat Postgres feature that allow reasonably efficient correlated subqueries. They are often described as SQL `for each` loops.

The desired SQL is:

```
SELECT * FROM
  (SELECT id, username FROM user) AS uq
   LEFT JOIN LATERAL
  (SELECT message, created_date
   FROM tweet
   WHERE (user_id = uq.id)
```

(continues on next page)

(continued from previous page)

```
   ORDER BY created_date DESC LIMIT 3)
  AS pq ON true
```

To accomplish this with peewee is quite straightforward:

```
subq = (Tweet
        .select(Tweet.message, Tweet.created_date)
        .where(Tweet.user == User.id)
        .order_by(Tweet.created_date.desc())
        .limit(3))

query = (User
         .select(User, subq.c.content, subq.c.created_date)
         .join(subq, JOIN.LEFT_LATERAL)
         .order_by(User.username, subq.c.created_date.desc()))

# We queried from the "perspective" of user, so the rows are User instances
# with the addition of a "content" and "created_date" attribute for each of
# the (up-to) 3 most-recent tweets for each user.
for row in query:
    print(row.username, row.content, row.created_date)
```

To implement an equivalent query from the "perspective" of the Tweet model, we can instead write:

```
# subq is the same as the above example.
subq = (Tweet
        .select(Tweet.message, Tweet.created_date)
        .where(Tweet.user == User.id)
        .order_by(Tweet.created_date.desc())
        .limit(3))

query = (Tweet
         .select(User.username, subq.c.content, subq.c.created_date)
         .from_(User)
         .join(subq, JOIN.LEFT_LATERAL)
         .order_by(User.username, subq.c.created_date.desc()))

# Each row is a "tweet" instance with an additional "username" attribute.
# This will print the (up-to) 3 most-recent tweets from each user.
for tweet in query:
    print(tweet.username, tweet.content, tweet.created_date)
```

Window functions

Window functions, which are *supported by peewee*, provide scalable, efficient performance.

The desired SQL is:

```
SELECT subq.message, subq.username
FROM (
    SELECT
        t2.message,
        t3.username,
        RANK() OVER (
            PARTITION BY t2.user_id
```

(continues on next page)

(continued from previous page)

```
            ORDER BY t2.created_date DESC
        ) AS rnk
    FROM tweet AS t2
    INNER JOIN user AS t3 ON (t2.user_id = t3.id)
) AS subq
WHERE (subq.rnk <= 3)
```

To accomplish this with peewee, we will wrap the ranked Tweets in an outer query that performs the filtering.

```
TweetAlias = Tweet.alias()

# The subquery will select the relevant data from the Tweet and
# User table, as well as ranking the tweets by user from newest
# to oldest.
subquery = (TweetAlias
            .select(
                TweetAlias.message,
                User.username,
                fn.RANK().over(
                    partition_by=[TweetAlias.user],
                    order_by=[TweetAlias.created_date.desc()]).alias('rnk'))
            .join(User, on=(TweetAlias.user == User.id))
            .alias('subq'))

# Since we can't filter on the rank, we are wrapping it in a query
# and performing the filtering in the outer query.
query = (Tweet
         .select(subquery.c.message, subquery.c.username)
         .from_(subquery)
         .where(subquery.c.rnk <= 3))
```

Other methods

If you're not using Postgres, then unfortunately you're left with options that exhibit less-than-ideal performance. For a more complete overview of common methods, check out this blog post. Below I will summarize the approaches and the corresponding SQL.

Using `COUNT`, we can get all tweets where there exist less than *N* tweets with more recent timestamps:

```
TweetAlias = Tweet.alias()

# Create a correlated subquery that calculates the number of
# tweets with a higher (newer) timestamp than the tweet we're
# looking at in the outer query.
subquery = (TweetAlias
            .select(fn.COUNT(TweetAlias.id))
            .where(
                (TweetAlias.created_date >= Tweet.created_date) &
                (TweetAlias.user == Tweet.user)))

# Wrap the subquery and filter on the count.
query = (Tweet
         .select(Tweet, User)
         .join(User)
         .where(subquery <= 3))
```

We can achieve similar results by doing a self-join and performing the filtering in the `HAVING` clause:

```
TweetAlias = Tweet.alias()

# Use a self-join and join predicates to count the number of
# newer tweets.
query = (Tweet
         .select(Tweet.id, Tweet.message, Tweet.user, User.username)
         .join(User)
         .switch(Tweet)
         .join(TweetAlias, on=(
             (TweetAlias.user == Tweet.user) &
             (TweetAlias.created_date >= Tweet.created_date)))
         .group_by(Tweet.id, Tweet.content, Tweet.user, User.username)
         .having(fn.COUNT(Tweet.id) <= 3))
```

The last example uses a `LIMIT` clause in a correlated subquery.

```
TweetAlias = Tweet.alias()

# The subquery here will calculate, for the user who created the
# tweet in the outer loop, the three newest tweets. The expression
# will evaluate to `True` if the outer-loop tweet is in the set of
# tweets represented by the inner query.
query = (Tweet
         .select(Tweet, User)
         .join(User)
         .where(Tweet.id << (
             TweetAlias
             .select(TweetAlias.id)
             .where(TweetAlias.user == Tweet.user)
             .order_by(TweetAlias.created_date.desc())
             .limit(3))))
```

1.16.4 Writing custom functions with SQLite

SQLite is very easy to extend with custom functions written in Python, that are then callable from your SQL statements. By using the *SqliteExtDatabase* and the `func()` decorator, you can very easily define your own functions.

Here is an example function that generates a hashed version of a user-supplied password. We can also use this to implement `login` functionality for matching a user and password.

```
from hashlib import sha1
from random import random
from playhouse.sqlite_ext import SqliteExtDatabase

db = SqliteExtDatabase('my-blog.db')

def get_hexdigest(salt, raw_password):
    data = salt + raw_password
    return sha1(data.encode('utf8')).hexdigest()

@db.func()
def make_password(raw_password):
    salt = get_hexdigest(str(random()), str(random()))[:5]
    hsh = get_hexdigest(salt, raw_password)
```

(continues on next page)

(continued from previous page)

```
    return '%s$%s' % (salt, hsh)

@db.func()
def check_password(raw_password, enc_password):
    salt, hsh = enc_password.split('$', 1)
    return hsh == get_hexdigest(salt, raw_password)
```

Here is how you can use the function to add a new user, storing a hashed password:

```
query = User.insert(
    username='charlie',
    password=fn.make_password('testing')).execute()
```

If we retrieve the user from the database, the password that's stored is hashed and salted:

```
>>> user = User.get(User.username == 'charlie')
>>> print user.password
b76fa$88be1adcde66a1ac16054bc17c8a297523170949
```

To implement `login`-type functionality, you could write something like this:

```
def login(username, password):
    try:
        return (User
                .select()
                .where(
                    (User.username == username) &
                    (fn.check_password(password, User.password) == True))
                .get())
    except User.DoesNotExist:
        # Incorrect username and/or password.
        return False
```

1.16.5 Date math

Each of the databases supported by Peewee implement their own set of functions and semantics for date/time arithmetic.

This section will provide a short scenario and example code demonstrating how you might utilize Peewee to do dynamic date manipulation in SQL.

Scenario: we need to run certain tasks every *X* seconds, and both the task intervals and the task themselves are defined in the database. We need to write some code that will tell us which tasks we should run at a given time:

```
class Schedule(Model):
    interval = IntegerField()  # Run this schedule every X seconds.


class Task(Model):
    schedule = ForeignKeyField(Schedule, backref='tasks')
    command = TextField()  # Run this command.
    last_run = DateTimeField()  # When was this run last?
```

Our logic will essentially boil down to:

```
.. code-block:: python
```

e.g., if the task was last run at 12:00:05, and the associated interval # is 10 seconds, the next occurrence should be 12:00:15. So we check # whether the current time (now) is 12:00:15 or later. now >= task.last_run + schedule.interval

So we can write the following code:

```
next_occurrence = something  # ??? how do we define this ???

# We can express the current time as a Python datetime value, or we could
# alternatively use the appropriate SQL function/name.
now = Value(datetime.datetime.now())  # Or SQL('current_timestamp'), e.g.

query = (Task
         .select(Task, Schedule)
         .join(Schedule)
         .where(now >= next_occurrence))
```

For Postgresql we will multiple a static 1-second interval to calculate the offsets dynamically:

```
second = SQL("INTERVAL '1 second'")
next_occurrence = Task.last_run + (Schedule.interval * second)
```

For MySQL we can reference the schedule's interval directly:

```
from peewee import NodeList  # Needed to construct sql entity.

interval = NodeList((SQL('INTERVAL'), Schedule.interval, SQL('SECOND')))
next_occurrence = fn.date_add(Task.last_run, interval)
```

For SQLite, things are slightly tricky because SQLite does not have a dedicated datetime type. So for SQLite, we convert to a unix timestamp, add the schedule seconds, then convert back to a comparable datetime representation:

```
next_ts = fn.strftime('%s', Task.last_run) + Schedule.interval
next_occurrence = fn.datetime(next_ts, 'unixepoch')
```

1.17 Changes in 3.0

This document describes changes to be aware of when switching from 2.x to 3.x.

1.17.1 Backwards-incompatible

I tried to keep changes backwards-compatible as much as possible. In some places, APIs that have changed will trigger a `DeprecationWarning`.

Database

- `get_conn()` has changed to *`Database.connection()`*
- `get_cursor()` has changed to *`Database.cursor()`*
- `execution_context()` is replaced by simply using the database instance as a context-manager.

- For a connection context *without* a transaction, use *Database.connection_context()*.
- *Database.create_tables()* and *Database.drop_tables()*, as well as *Model.create_table()* and *Model.drop_table()* all default to safe=True (create_table will create if not exists, drop_table will drop if exists).
- connect_kwargs attribute has been renamed to connect_params
- initialization parameter for custom field-type definitions has changed from fields to field_types.

Model Meta options

- db_table has changed to table_name
- db_table_func has changed to table_function
- order_by has been removed (used for specifying a default ordering to be applied to SELECT queries).
- validate_backrefs has been removed. Back-references are no longer validated.

Models

- BaseModel has been renamed to ModelBase
- Accessing raw model data is now done using __data__ instead of _data
- The _prepare_instance() Model method has been removed.
- The sqlall() method, which output the DDL statements to generate a model and its associated indexes, has been removed.

Fields

- db_column has changed to column_name
- db_field class attribute changed to field_type (used if you are implementing custom field subclasses)
- model_class attribute has changed to model
- PrimaryKeyField has been renamed to *AutoField*
- *ForeignKeyField* constructor has the following changes:
  - rel_model has changed to model
  - to_field has changed to field
  - related_name has changed to backref
- *ManyToManyField* is now included in the main peewee.py module
- Removed the extension fields PasswordField, PickledField and AESEncryptedField.

Querying

JOIN_INNER, JOIN_LEFT_OUTER, etc are now JOIN.INNER, JOIN.LEFT_OUTER, etc.

The C extension that contained implementations of the query result wrappers has been removed.

Additionally, Select.aggregate_rows() has been removed. This helper was used to de-duplicate left-join queries to give the appearance of efficiency when iterating a model and its relations. In practice, the complexity of the

code and its somewhat limited usefulness convinced me to scrap it. You can instead use *prefetch()* to achieve the same result.

- *Select* query attribute _select has changed to _returning
- The naive() method is now *objects()*, which defaults to using the model class as the constructor, but accepts any callable to use as an alternate constructor.
- The annotate() query method is no longer supported.

The *Case()* helper has moved from the playhouse.shortcuts module into the main peewee module.

The cast() method is no longer a function, but instead is a method on all column-like objects.

The InsertQuery.return_id_list() method has been replaced by a more general pattern of using *_WriteQuery.returning()*.

The InsertQuery.upsert() method has been replaced by the more general and flexible *Insert.on_conflict()* method.

When using *prefetch()*, the collected instances will be stored in the same attribute as the foreign-key's backref. Previously, you would access joined instances using (backref)_prefetch.

The *SQL* object, used to create a composable a SQL string, now expects the second parameter to be a list/tuple of parameters.

Removed Extensions

The following extensions are no longer included in the playhouse:

- berkeleydb
- csv_utils
- djpeewee
- gfk
- kv
- pskel
- read_slave

SQLite Extension

The SQLite extension module's *VirtualModel* class accepts slightly different Meta options:

- arguments - used to specify arbitrary arguments appended after any columns being defined on the virtual table. Should be a list of strings.
- extension_module (unchanged)
- options (replaces extension_options) - arbitrary options for the virtual table that appear after columns and arguments.
- prefix_arguments - a list of strings that should appear before any arguments or columns in the virtual table declaration.

So, when declaring a model for a virtual table, it will be constructed roughly like this:

```
CREATE VIRTUAL TABLE "table name" USING extension_module (
    prefix arguments,
    field definitions,
    arguments,
    options)
```

Postgresql Extension

The *PostgresqlExtDatabase* no longer registers the *hstore* extension by default. To use the *hstore* extension in 3.0 and onwards, pass *register_hstore=True* when initializing the database object.

Signals Extension

The `post_init` signal has been removed.

1.17.2 New stuff

The query-builder has been rewritten from the ground-up to be more flexible and powerful. There is now a generic, *lower-level API* for constructing queries.

SQLite

Many SQLite-specific features have been moved from the `playhouse.sqlite_ext` module into `peewee`, such as:

- User-defined functions, aggregates, collations, and table-functions.
- Loading extensions.
- Specifying pragmas.

See the *"Using SQLite" section* and *"SQLite extensions"* documents for more details.

SQLite Extension

The virtual-table implementation from sqlite-vtfunc has been folded into the peewee codebase.

- Support for SQLite online backup API.
- Murmurhash implementation has been corrected.
- Couple small quirks in the BM25 ranking code have been addressed.
- Numerous user-defined functions for hashing and ranking are now included.
- `BloomFilter` implementation.
- Incremental `Blob` I/O support.
- Support for update, commit and rollback hooks.
- `LSMTable` implementation to support the lsm1 extension.

CHAPTER 2

Note

If you find any bugs, odd behavior, or have an idea for a new feature please don't hesitate to open an issue on GitHub or contact me.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

S

T

U

V

W

Y

Z