peewee Documentation

Release 2.10.1

charles leifer

Aug 17, 2017

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
- Written in python with support for versions 2.6+ and 3.2+.
- built-in support for sqlite, mysql and postgresql
- *numerous extensions available* (*postgres hstore/json/arrays*, *sqlite full-text-search*, *schema migrations*, and much more).

Peewee's source code hosted on GitHub.

New to peewee? Here is a list of documents you might find most helpful when getting started:

- *Quickstart guide* – this guide covers all the bare essentials. It will take you between 5 and 10 minutes to go through it.
- *Guide to the various query operators* describes how to construct queries and combine expressions.
- *Field types table* lists the various field types peewee supports and the parameters they accept. There is also an *extension module* that contains *special/custom field types*.

CHAPTER 1

Contents:

Installing and Testing

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

```
pip install peewee
```

Peewee comes with two C extensions that can optionally be compiled:

- Speedups, which includes miscellaneous functions re-implemented with Cython. This module will be built automatically if Cython is installed.
- Sqlite extensions, which includes Cython implementations of the SQLite date manipulation functions, the REGEXP operator, and full-text search result ranking algorithms. This module should be built using the `build_sqlite_ext` command.

Note: If you have Cython installed, then the `speedups` module will automatically be built. If you wish to also build the SQLite Cython extension, you must manually run:

```
python setup.py build_sqlite_ext
python setup.py install
```

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
```

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

```
# Build the sqlite extension and place the shared library alongside the other modules.
python setup.py build_sqlite_ext -i
```

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

Running tests

You can test your installation by running the test suite.

```
python setup.py test

# Or use the test runner:
python runtests.py
```

You can test specific features or specific database drivers using the `runtests.py` script. By default the test suite is run using SQLite and the `playhouse` extension tests are not run. To view the available test runner options, use:

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

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 for various speedups. Can give a big boost to certain operations, particularly if you use SQLite.
- apsw: an optional 3rd-party SQLite binding offering greater performance and much, much saner semantics than the standard library `pysqlite`. Use with *`APSWDatabase`*.
- pycrypto is used for the `AESEncryptedField`.
- `bcrypt` module is used for the *`PasswordField`*.
- *vtfunc <https://github.com/coleifer/sqlite-vtfunc>* is used to provide some table-valued functions for Sqlite as part of the `sqlite_udf` extensions module.
- 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* or *Django* frameworks, there are helper extension modules available.

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.

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

Model Definition

Model classes, fields 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

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()
    is_relative = BooleanField()

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

Note: 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 keys (wikipedia). This is easy to do with peewee:

```
class Pet(Model):
    owner = ForeignKeyField(Person, related_name='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])
```

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), is_relative=True)
>>> uncle_bob.save() # bob is now stored in the database
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), is_
↪relative=True)
>>> herb = Person.create(name='Herb', birthday=date(1950, 5, 5), is_relative=False)
```

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.
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
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()
>>> bob_fido = herb_fido # rename our variable for clarity
```

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 *SelectQuery.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, person.is_relative
...
Bob True
Grandma L. True
Herb False
```

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
...
Kitty Bob
Mittens Jr Herb
```

There is a big problem with the previous query: because we are accessing `pet.owner.name` and we did not select this value 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.

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
...
```

```
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
...
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
```

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

Now let's list all the people *and* some info about their pets:

```
>>> for person in Person.select():
...     print person.name, person.pets.count(), 'pets'
...     for pet in person.pets:
...         print '    ', pet.name, pet.animal_type
...
Bob 2 pets
    Kitty cat
    Fido dog
Grandma L. 0 pets
Herb 1 pets
    Mittens Jr cat
```

Once again we've run into a classic example of *N+1* query behavior. We can avoid this by performing a *JOIN* and aggregating the records:

```
>>> subquery = Pet.select(fn.COUNT(Pet.id)).where(Pet.owner == Person.id)
>>> query = (Person
...          .select(Person, Pet, subquery.alias('pet_count'))
...          .join(Pet, JOIN.LEFT_OUTER)
...          .order_by(Person.name))

>>> for person in query.aggregate_rows():  # Note the `aggregate_rows()` call.
```

```
...     print person.name, person.pet_count, 'pets'
...     for pet in person.pets:
...         print '    ', pet.name, pet.animal_type
...
Bob 2 pets
     Kitty cat
     Fido dog
Grandma L. 0 pets
Herb 1 pets
     Mittens Jr cat
```

Even though we created the subquery separately, **only one** query is actually executed.

Finally, let's do a complicated one. 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
...
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 > d1940) & (Person.birthday < d1960)))
...
>>> for person in query:
...     print person.name, person.birthday
...
Herb 1950-05-05
```

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
...
Grandma L.
```

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

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

This is just the basics! You can make your queries as complex as you like.

All the other SQL clauses are available as well, such as:

- *group_by()*

- *having()*
- *limit()* and *offset()*

Check the documentation on *Querying* for more info.

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
```

What next?

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

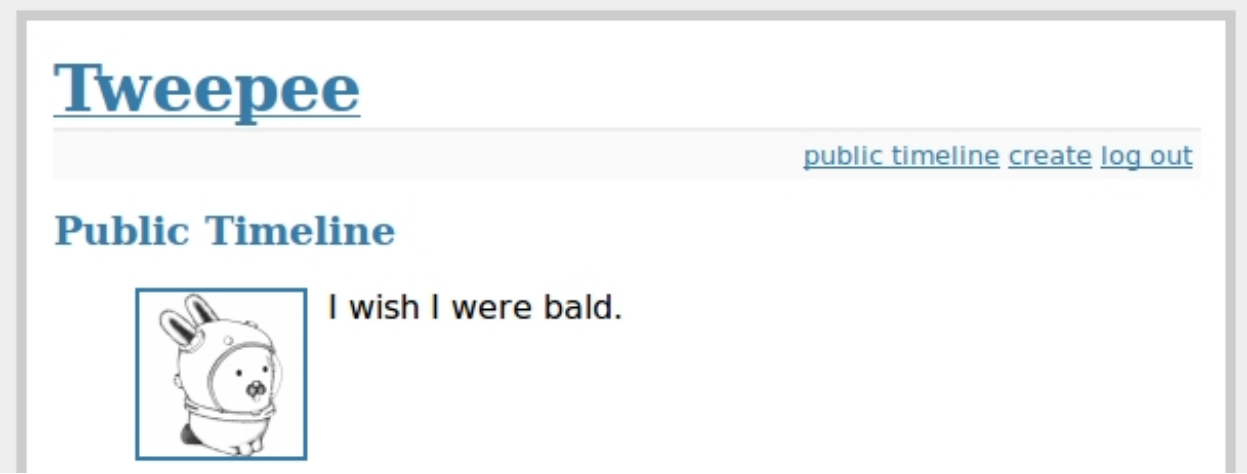
Example app

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.

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
```

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/

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

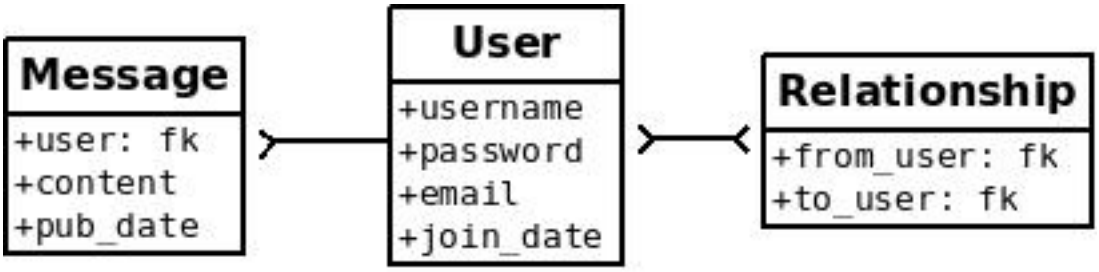
In the spirit of the popular web framework Django, peewee uses declarative model definitions. If you're not familiar with Django, the idea is that you declare a model class for each table. The model class then defines one or more field attributes which correspond to the table's columns. For the twitter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

    class Meta:
        order_by = ('username',)

# 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, related_name='relationships')
    to_user = ForeignKeyField(User, related_name='related_to')

    class Meta:
        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.message_set
class Message(BaseModel):
    user = ForeignKeyField(User)
    content = TextField()
    pub_date = DateTimeField()

    class Meta:
        order_by = ('-pub_date',)
```

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():
    database.connect()
    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.

Note: Adding fields after the table has been created will required 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.

Note: You can also write `database.create_tables([User, ...], True)` and peewee will first check to see if the table exists before creating it.

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, threadlocals=True)
```

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))

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

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.transaction():
        # 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.transaction():
        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:

```
# python code
messages = Message.select().where(Message.user << user.following())
```

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)
```

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.

Additional Resources

I've written a number of blog posts about building applications and web-services with peewee (and usually Flask). If you'd like to see some "real-life" applications that use peewee, the following resources may be useful:

- How to make a Flask blog in one hour or less.
- Building a note-taking app with Flask and Peewee as well as Part 2 and Part 3.
- Analytics web service built with Flask and Peewee.
- Personalized news digest (with a boolean query parser!).
- Using peewee to explore CSV files.
- Structuring Flask apps with Peewee.
- Creating a lastpass clone with Flask and Peewee.
- Building a web-based encrypted file manager with Flask, peewee and S3.
- Creating a bookmarking web-service that takes screenshots of your bookmarks.
- Building a pastebin, wiki and a bookmarking service using Flask and Peewee.
- Encrypted databases with Python and SQLCipher.
- Dear Diary, an Encrypted Command-Line Diary.

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.

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.

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!

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 in IRC, `#peewee` on freenode. I always answer questions, but it may take a bit to get to them.
- Ask on the mailing list, https://groups.google.com/group/peewee-orm

Managing your Database

This document describes how to perform typical database-related tasks with peewee. Throughout this document we will use the following example models:

```
from peewee import *

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

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

Creating a database connection and tables

While it is not necessary to explicitly connect to the database before using it, **managing connections explicitly is a good practice**. This way if the connection fails, the exception can be caught during the *connect* step, rather than some arbitrary time later when a query is executed. Furthermore, if you're using a *connection pool*, it is actually necessary to call `connect()` and `close()` to ensure connections are recycled correctly.

For web-apps you will typically open a connection when a request is started and close it when the response is delivered:

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

def before_request_handler():
    database.connect()

def after_request_handler():
    database.close()
```

Note: For examples of configuring connection hooks for several popular web frameworks, see the *Adding Request Hooks* section.

Note: For advanced connection management techniques, see the *advanced connection management* section.

To use this database with your models, set the `database` attribute on an inner *Meta* class:

```
class MyModel(Model):
    some_field = CharField()

    class Meta:
        database = database
```

Best practice: define a base model class that points at the database object you wish to use, and then all your models will extend it:

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

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

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

```
class Tweet(BaseModel):
    user = ForeignKeyField(User, related_name='tweets')
    message = TextField()
    # etc, etc
```

Note: Remember to specify a database on your model classes, otherwise peewee will fall back to a default sqlite database named "peewee.db".

Vendor-specific Parameters

Some database drivers accept special parameters when being initialized. Rather than try to accommodate all these parameters, Peewee will pass back unrecognized parameters directly to the database driver.

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

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')
```

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 arbitrary sqlite3 parameters.

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

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

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

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

- *Full-text search* with *BM25 ranking*.
- Support for custom functions, aggregates and collations
- Advanced transaction support
- And 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', journal_mode='WAL')
```

PRAGMA statements

New in version 2.6.4.

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

```
db = SqliteDatabase('my_app.db', pragmas=(
    ('journal_mode', 'WAL'),
    ('cache_size', 10000),
    ('mmap_size', 1024 * 1024 * 32),
))
```

SQLite and Autocommit

Changed in version 2.4.5.

In version 2.4.5, the default isolation level for SQLite databases is `None`, which equates to *autocommit*. The reason for this change has to do with some idiosyncracies of `pysqlite` (or the standard library `sqlite3`).

If you are using your database in autocommit mode (the default) then you should not need to make any changes to your code.

If you are using `autocommit=False`, you will need to explicitly call *`begin()`* before executing queries.

Note: This does not apply to code executed within *`transaction()`* or *`atomic()`*.

Warning: If you are using peewee with autocommit disabled, you must explicitly call *`begin()`*, otherwise statements **will** be executed in autocommit mode.

Example code:

```
# Define a database with autocommit turned off.
db = SqliteDatabase('my_app.db', autocommit=False)

# You must call begin()
db.begin()
User.create(username='charlie')
db.commit()

# If using a transaction, then no changes are necessary.
with db.transaction():
    User.create(username='huey')

# If using a function decorated by transaction, no changes are necessary.
@db.transaction()
def create_user(username):
    User.create(username=username)
```

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')
```

Using BerkeleyDB

The *playhouse* contains a special extension module for using a *BerkeleyDB database*. BerkeleyDB can be compiled with a SQLite-compatible API, then the python SQLite driver can be compiled to use the Berkeley version of SQLite.

You can find up-to-date step by step instructions on my blog for compling the BerkeleyDB + SQLite library, then building a statically-linked pysqlite that uses the custom sqlite library.

To connect to a BerkeleyDB database, we will use *BerkeleyDatabase*. Like *SqliteDatabase*, the first parameter is the filename containing the database or the string *:memory:* to create an in-memory database.

```
from playhouse.berkeleydb import BerkeleyDatabase

berkeley_db = BerkeleyDatabase('my_app.db')

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

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

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 *Adding Request Hooks* for more information.

If you would like to automatically reconnect and retry queries that fail due to an `OperationalError`, peewee provides a `Database` mixin `RetryOperationalError` that will handle reconnecting and retrying the query automatically. For more information see *Automatic Reconnect*.

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*.

Multi-threaded applications

peewee stores the connection state in a thread local, so each thread gets its own separate connection. If you prefer to manage the connections yourself, you can disable this behavior by initializing your database with `threadlocals=False`.

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 = SqliteDatabase(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 = raw_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*.

Dynamically defining a database

For even more control over how your database is defined/initialized, you can use the `Proxy` helper. `Proxy` 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 = Proxy()  # 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 `Proxy` 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.

Connection Pooling

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

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

The connection pool module comes with support for Postgres and MySQL (though adding support for other databases is trivial).

```
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, extensions to Peewee* documentation.

Read Slaves

Peewee can automatically run *SELECT* queries against one or more read replicas. The *read_slave module*, included in the *Playhouse, extensions to Peewee* extensions library, contains a `Model` subclass which provides this behavior.

Here is how you might use the `ReadSlaveModel`:

```
from peewee import *
from playhouse.read_slave import ReadSlaveModel

# Declare a master and two read-replicas.
master = PostgresqlDatabase('master')
replica_1 = PostgresqlDatabase('replica', host='192.168.1.2')
replica_2 = PostgresqlDatabase('replica', host='192.168.1.3')

class BaseModel(ReadSlaveModel):
```

```
    class Meta:
        database = master
        read_slaves = (replica_1, replica_2)

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

Now when you execute writes (or deletes), they will be run on the master, while all read-only queries will be executed against one of the replicas. Queries are dispatched among the read slaves in round-robin fashion.

Schema migrations

Currently peewee does not have support for *automatic* schema migrations, but you can use the *Schema Migrations* module to create simple migration scripts. The schema migrations module works with SQLite, MySQL and Postgres, and will even allow you to do things like drop or rename columns in SQLite!

Here is an example of how you might write a migration script:

```
from playhouse.migrate import *

my_db = SqliteDatabase('my_database.db')
migrator = SqliteMigrator(my_db)

title_field = CharField(default='')
status_field = IntegerField(null=True)

with my_db.transaction():
    migrate(
        migrator.add_column('some_table', 'title', title_field),
        migrator.add_column('some_table', 'status', status_field),
        migrator.drop_column('some_table', 'old_column'),
    )
```

Check the *Schema Migrations* documentation for more details.

Generating Models from Existing Databases

If you'd like to generate peewee model definitions for an existing database, you can try out the database introspection tool *pwiz, a model generator* that comes with peewee. *pwiz* is capable of introspecting Postgresql, MySQL and SQLite databases.

Introspecting a Postgresql database:

```
python -m pwiz --engine=postgresql my_postgresql_database
```

Introspecting a SQLite database:

```
python -m pwiz --engine=sqlite test.db
```

pwiz will generate:

- Database connection object
- A *BaseModel* class to use with the database
- *Model* classes for each table in the database.

The generated code is written to stdout, and can easily be redirected to a file:

```
python -m pwiz -e postgresql my_postgresql_db > models.py
```

Note: pwiz generally works quite well with even large and complex database schemas, but in some cases it will not be able to introspect a column. You may need to go through the generated code to add indexes, fix unrecognized column types, and resolve any circular references that were found.

Adding Request Hooks

When building web-applications, it is very important that you manage your database connections correctly. 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.

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.


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 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()

    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)
```

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.

Additional connection initialization

Peewee does a few basic things depending on your database to initialize a connection. For SQLite this means registering custom user-defined functions, for Postgresql this means registering unicode support.

You may find it necessary to add additional initialization when a new connection is opened, however. For example you may want to tell SQLite to enforce all foreign key constraints (off by default). To do this, you can subclass the database and override the *initialize_connection()* method.

This method contains no implementation on the base database classes, so you do not need to call `super()` with it.

Example turning on SQLite foreign keys:

```
class SqliteFKDatabase(SqliteDatabase):
    def initialize_connection(self, conn):
        self.execute_sql('PRAGMA foreign_keys=ON;')
```

Advanced Connection Management

Managing your database connections is as simple as calling *connect()* when you need to open a connection, and *close()* when you are finished. In a web-app, you would typically connect when you receive a request, and close the connection when you return a response. Because connection state is stored in a thread-local, you do not need to worry about juggling connection objects – peewee will handle it for you.

In some situations, however, you may want to manage your connections more explicitly. Since peewee stores the active connection in a threadlocal, this typically would mean that there could only ever be one connection open per thread. For most applications this is desirable, but if you would like to manually manage multiple connections you can create an *ExecutionContext*.

Execution contexts allow finer-grained control over managing multiple connections to the database. When an execution context is initialized (either as a context manager or as a decorated function), a separate connection will be used for the duration of the wrapped block. You can also choose whether to wrap the block in a transaction.

Execution context examples:

```
with db.execution_context() as ctx:
    # A new connection will be opened or, if using a connection pool,
    # pulled from the pool of available connections. Additionally, a
    # transaction will be started.
    user = User.create(username='charlie')
```

```
# When the block ends, the transaction will be committed and the connection
# will be closed (or returned to the pool).

@db.execution_context(with_transaction=False)
def do_something(foo, bar):
    # When this function is called, a separate connection is made and will
    # be closed when the function returns.
```

If you are using the peewee connection pool, then the new connections used by the *ExecutionContext* will be pulled from the pool of available connections and recycled appropriately.

Using multiple databases

With peewee you can use as many databases as you want. Each model can define it's database by specifying a *Meta.database*. What if you want to use the same model with multiple databases, though? Depending on your use-case, peewee provides several options.

If you have a Master/Slave setup and want all writes to go to the master, but reads can go to any number of replicated copies, check out the *Read Slave extension*.

For finer-grained control, check out the *Using* context manager / decorator. This allows you to specify the database to use with a given list of models for the duration of the wrapped block.

Here is an example of how you might use the *Using* context manager:

```
master = PostgresqlDatabase('master')
read_replica = PostgresqlDatabase('replica')

class Data(Model):
    value = IntegerField()

    class Meta:
        database = master

# By default all queries go to the master, since that is what
# is defined on our model.
for i in range(10):
    Data.create(value=i)

# But what if we want to explicitly use the read replica?
with Using(read_replica, [Data]):
    # Query is executed against the read replica.
    Data.get(Data.value == 5)

    # Since we did not specify this model in the list of overrides
    # it will use whatever database it was defined with.
    SomeOtherModel.get(SomeOtherModel.field == 3)
```

Note: For simple master/slave configurations, check out the *Read Slaves* extension. This extension ensures writes are sent to the master database and reads occur from any of the listed read replicas.

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`.

Automatic Reconnect

Peewee provides very basic support for automatic reconnecting in the *Shortcuts* module, through the use of the *`RetryOperationalError`* mixin. This mixin will automatically reconnect to the database and retry any queries that fail with an `OperationalError`. The query that failed will be retried only once, and if it fails twice an exception will be raised.

Usage:

```
from peewee import *
from playhouse.shortcuts import RetryOperationalError


class MyRetryDB(RetryOperationalError, MySQLDatabase):
    pass


db = MyRetryDB('my_app')
```

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.setLevel(logging.DEBUG)
logger.addHandler(logging.StreamHandler())
```

Generating skeleton code

For writing quick scripts, peewee comes with a helper script *pskel* which generates database connection and model boilerplate code. If you find yourself frequently writing small programs, *pskel* can really save you time.

To generate a script, you can simply run:

```
pskel User Tweet SomeModel AnotherModel > my_script.py
```

`pskel` will generate code to connect to an in-memory SQLite database, as well as blank model definitions for the model names specified on the command line.

Here is a more complete example, which will use the *PostgresqlExtDatabase* with query logging enabled:

```
pskel -l -e postgres_ext -d my_database User Tweet > my_script.py
```

You can now fill in the model definitions and get to hacking!

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()*
- interpolation and quote_char
- op_overrides 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()
```

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.

```
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, related_name='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.

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()
```

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, related_name='messages')
    body = TextField()
    send_date = DateTimeField()
```

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
```

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

Field types table

Field Type	Sqlite	Postgresql	MySQL
`CharField`	varchar	varchar	varchar
`FixedCharField`	char	char	char
`TextField`	text	text	longtext
`DateTimeField`	datetime	timestamp	datetime
`IntegerField`	integer	integer	integer
`BooleanField`	integer	boolean	bool
`FloatField`	real	real	real
`DoubleField`	real	double precision	double precision
`BigIntegerField`	integer	bigint	bigint
`SmallIntegerField`	integer	smallint	smallint
`DecimalField`	decimal	numeric	numeric
`PrimaryKeyField`	integer	serial	integer
`ForeignKeyField`	integer	integer	integer
`DateField`	date	date	date
`TimeField`	time	time	time
`TimestampField`	integer	integer	integer
`BlobField`	blob	bytea	blob
`UUIDField`	text	uuid	varchar(40)
`BareField`	untyped	not supported	not supported

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` – boolean indicating whether null values are allowed to be stored
- `index = False` – boolean indicating whether to create an index on this column
- `unique = False` – boolean indicating whether to create a unique index on this column. See also *adding composite indexes*.
- `verbose_name = None` – string representing the "user-friendly" name of this field
- `help_text = None` – string representing any helpful text for this field
- `db_column = None` – string representing the underlying column to use if different, useful for legacy databases
- `default = None` – any value to use as a default for uninitialized models; If `callable`, will be called to produce value
- `choices = None` – an optional iterable containing 2-tuples of `value`, `display`
- `primary_key = False` – whether this field is the primary key for the table
- `sequence = None` – sequence to populate field (if backend supports it)
- `constraints = None` - a list of one or more constraints, e.g. `[Check('price > 0')]`
- `schema = None` – optional name of the schema to use, if your db supports this.

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	rel_model, related_name, to_field, on_delete, on_update, extra
BareField	coerce

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 `to_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.

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

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

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.create_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)
```

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 via the parameter `related_name`:

```
class Message(Model):
    from_user = ForeignKeyField(User)
    to_user = ForeignKeyField(User, related_name='received_messages')
    text = TextField()

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

for message in some_user.received_messages:
    # 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`.

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.

BareField accepts a special parameter `coerce`. 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 `coerce=int`.

Creating a custom field

It isn't too difficult 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.db_field` attribute.

Let's start by defining our UUID field:

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

    def db_value(self, value):
        return str(value) # convert UUID to str

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

Now, we need to let the database know how to map this *uuid* label to an actual *uuid* column type in the database. There are 2 ways of doing this:

1. Specify the overrides in the *Database* constructor:

```
db = PostgresqlDatabase('my_db', fields={'uuid': 'uuid'})
```

2. Register them class-wide using *Database.register_fields()*:

```
# Will affect all instances of PostgresqlDatabase
PostgresqlDatabase.register_fields({'uuid': 'uuid'})
```

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`.

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: Peewee can determine if your tables already exist, and conditionally create them:

```
# Only create the tables if they do not exist.
db.create_tables([User, Tweet], safe=True)
```

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.

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.PrimaryKeyField object at 0x7f51a2e92750>, 'name': <peewee.CharField␣
↪object at 0x7f51a2f0a510>}

>>> Person._meta.primary_key
<peewee.PrimaryKeyField 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
`db_table`	name of the table to store data	no
`db_table_func`	function that accepts model and returns a table name	yes
`indexes`	a list of fields to index	yes
`order_by`	a list of fields to use for default ordering	yes
`primary_key`	a *`CompositeKey`* instance	yes
`table_alias`	an alias to use for the table in queries	no
`schema`	the database schema for the model	yes
`constraints`	a list of table constraints	yes
`validate_backrefs`	ensure backrefs do not conflict with other attributes.	yes
`only_save_dirty`	when calling model.save(), only save dirty fields	yes

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

```
>>> db = SqliteDatabase(':memory:')
>>> class ModelOne(Model):
...     class Meta:
```

```
...         database = db
...         db_table = 'model_one_tbl'
...
>>> class ModelTwo(ModelOne):
...     pass
...
>>> ModelOne._meta.database is ModelTwo._meta.database
True
>>> ModelOne._meta.db_table == ModelTwo._meta.db_table
False
```

Meta.order_by

Specifying a default ordering is, in my opinion, a bad idea. It's better to be explicit in your code when you want to sort your results.

That said, to specify a default ordering, the syntax is similar to that of Django. `Meta.order_by` is a tuple of field names, and to indicate descending ordering, the field name is prefixed by a `'-'`.

```
class Person(Model):
    first_name = CharField()
    last_name = CharField()
    dob = DateField()

    class Meta:
        # Order people by last name, first name. If two people have the
        # same first and last, order them youngest to oldest.
        order_by = ('last_name', 'first_name', '-dob')
```

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
```

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 are 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
            (('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!
    )
```

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')]
```

Non-integer Primary Keys, Composite Keys and other Tricks

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')
```

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.transaction():
    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
```

If you *always* want to have control over the primary key, simply do not use the *PrimaryKeyField* 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 *~Model.delete_instance* (you can instead use *~Model.insert*, *~Model.update* and *~Model.delete*).

Self-referential foreign keys

When creating a heirarchical 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, related_name='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))
```

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, related_name='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 *DeferredRelation* we can get around the problem and still use a foreign key field:

```
# Create a reference object to stand in for our as-yet-undefined Tweet model.
DeferredTweet = DeferredRelation()

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

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

# Now that Tweet is defined, we can initialize the reference.
DeferredTweet.set_model(Tweet)
```

After initializing the deferred relation, the foreign key fields are now correctly set up. 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.

Here is how to create the tables:

```
# Foreign key constraint from User -> Tweet will NOT be created because the
# Tweet table does not exist yet. `favorite_tweet` will just be a regular
# integer field:
User.create_table()

# Foreign key constraint from Tweet -> User will be created normally.
Tweet.create_table()

# Now that both tables exist, we can create the foreign key from User -> Tweet:
```

```
# NOTE: this will not work in SQLite!
db.create_foreign_key(User, User.favorite_tweet)
```

Warning: SQLite does not support adding constraints to existing tables through the `ALTER TABLE` statement.

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.

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
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.

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:
    Model.create(**data_dict)
```

The above approach is slow for a couple of reasons:

1. If you are using autocommit (the default), 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 **very significant speedup** by simply wrapping this in a *atomic()*.

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

The above code still suffers from points 2, 3 and 4. We can get another big boost by calling *insert_many()*. This method accepts a list of dictionaries to insert.

```
# Fastest.
with db.atomic():
    Model.insert_many(data_source).execute()
```

Depending on the number of rows in your data source, you may need to break it up into chunks:

```
# Insert rows 100 at a time.
with db.atomic():
    for idx in range(0, len(data_source), 100):
        Model.insert_many(data_source[idx:idx+100]).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`. This value can be modified by setting the `SQLITE_MAX_VARIABLE_NUMBER` flag.

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:

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

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()* and *UpdateQuery*.

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.

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()
```

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*

Selecting a single record

You can use the *Model.get()* method to retrieve a single instance matching the given query.

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(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 *SelectQuery.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.select()*
- *SelectQuery.get()*

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 check out the documentation for *Model.get_or_create()*.

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.

In the following example, we will simply call *select()* and iterate over the return value, which is an instance of *SelectQuery*. 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 *SelectQuery.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 *related_name* to create a back-reference (`User.tweets`). Back-references are exposed as *SelectQuery* 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.
<SelectQuery> SELECT t1."id", t1."user_id", t1."message", t1."created_date", t1."is_
↪published" FROM "tweet" AS t1 WHERE (t1."user_id" = ?) [1]
```

You can iterate over the `user.tweets` back-reference just like any other *SelectQuery*:

```
>>> for tweet in user.tweets:
...     print tweet.message
...
hello world
this is fun
look at this picture of my food
```

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 '%s: %s' % (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')
... )
```

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 "<<" operator signifies an "IN" query
a_user_tweets = Tweet.select().where(Tweet.user << a_users)
```

More query examples

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 << staff_super)
```

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:
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:

```
>>> qry = 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)
         .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()))
```

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)
```

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
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()*.

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
```

In some cases it may be necessary to wrap your query and apply a count to the rows of the inner query (such as when using *DISTINCT* or *GROUP BY*). Peewee will usually do this automatically, but in some cases you may need to manually call *wrapped_count()* instead.

Aggregating records

Suppose you have some users and want to get a list of them along with the count of tweets in each. The *annotate()* method provides a short-hand for creating these types of queries:

```
query = User.select().annotate(Tweet)
```

The above query is equivalent to:

```
query = (User
         .select(User, fn.Count(Tweet.id).alias('count'))
         .join(Tweet)
         .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. By default it uses an inner join if the foreign key is not nullable, which means users without tweets won't appear in the list. To remedy this, manually specify the type of join to include users with 0 tweets:

```
query = (User
         .select()
         .join(Tweet, JOIN.LEFT_OUTER)
         .switch(User)
         .annotate(Tweet))
```

You can also specify a custom aggregator, such as *MIN* or *MAX*:

```
query = (User
         .select()
         .annotate(
             Tweet,
             fn.Max(Tweet.created_date).alias('latest_tweet_date')))
```

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)
         .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))
```

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)
```

SQL Functions, Subqueries and "Raw expressions"

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
query = User.select(User, fn.Lower(fn.Substr(User.username, 1, 1)).
↪alias('first_letter'))

# Alternatively we could select only users whose username begins with 'a'
a_users = User.select().where(fn.Lower(fn.Substr(User.username, 1, 1)) == 'a')

>>> for user in a_users:
...     print user.username
```

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'))
```

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*.

Example:

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

class Person(Model):
    name = CharField()
    class Meta:
        database = db

# let's pretend we want to do an "upsert", something that SQLite can
# do, but peewee cannot.
for name in ('charlie', 'mickey', 'huey'):
    db.execute_sql('REPLACE INTO person (name) VALUES (?)', (name,))

# now let's iterate over the people using our own query.
for person in Person.raw('select * from person'):
    print person.name  # .raw() will return model instances.
```

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!
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!
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.interpolation`.

Window functions

peewee comes with basic support for SQL window functions, which can be created by calling *fn.over()* and passing in your partitioning or ordering parameters.

```
# Get the list of employees and the average salary for their dept.
query = (Employee
         .select(
             Employee.name,
             Employee.department,
             Employee.salary,
             fn.Avg(Employee.salary).over(
                 partition_by=[Employee.department]))
         .order_by(Employee.name))
```

```
# Rank employees by salary.
query = (Employee
         .select(
             Employee.name,
             Employee.salary,
             fn.rank().over(
                 order_by=[Employee.salary])))
```

For general information on window functions, check out the postgresql docs.

Retrieving raw tuples / dictionaries

Sometimes you do not need the overhead of creating model instances and simply want to iterate over the row tuples. To do this, call *SelectQuery.tuples()* or *RawQuery.tuples()*:

```
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 *SelectQuery.dicts()* or *RawQuery.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']
```

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.

For example, let's say you have an *UpdateQuery* 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)
         .returning(User))

# Send an email to every user that was deactivated.
for deactivate_user in query.execute():
    send_deactivation_email(deactivated_user)
```

The RETURNING clause is also available on *InsertQuery* and *DeleteQuery*. 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.

For more information, see:

- *`UpdateQuery.returning()`*
- *`InsertQuery.returning()`*
- *`DeleteQuery.returning()`*

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
`~`	Negation

Because I ran out of operators to override, there are some additional query operations available as methods:

Method	Meaning
`.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.
`.bin_and(value)`	Binary AND.
`.bin_or(value)`	Binary OR.
`.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.
`.concat(other)`	Concatenate two strings using `||`.

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 << ['foo', 'bar', 'baz'])`

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 << ['charlie', 'huey', 'mickey'])
```

```
Employee.select().where(Employee.salary.between(50000, 60000))

Employee.select().where(Employee.name.startswith('C'))

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 << admins))

# Find any users who are not my friends (strangers).
friends = User.select().where(
  User.username << ['charlie', 'huey', 'mickey'])
strangers = User.select().where(~(User.id << 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 `<<` instead of `in`
- Use `&` instead of `and`
- Use `|` instead of `or`
- Use `~` instead of `not`
- 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.

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))
```

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

OP['MOD'] = 'mod'

def mod(lhs, rhs):
    return Expression(lhs, OP.MOD, rhs)

SqliteDatabase.register_ops({OP.MOD: '%'})
```

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.

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!

Foreign Keys

Foreign keys are created using a special field class *ForeignKeyField*. Each foreign key also creates a back-reference on the related model using the specified *related_name*.

Traversing foreign keys

Referring back to the *User and Tweet models*, note that there is a *ForeignKeyField* from *Tweet* to *User*. The foreign key can be traversed, allowing you access to the associated user instance:

```
>>> tweet.user.username
'charlie'
```

Note: Unless the *User* model was explicitly selected when retrieving the *Tweet*, an additional query will be required to load the *User* data. To learn how to avoid the extra query, see the *N+1 query documentation*.

The reverse is also true, and we can iterate over the tweets associated with a given *User* instance:

```
>>> for tweet in user.tweets:
...     print tweet.message
...
http://www.youtube.com/watch?v=xdhLQCYQ-nQ
```

Under the hood, the *tweets* attribute is just a *SelectQuery* with the *WHERE* clause pre-populated to point to the given *User* instance:

```
>>> user.tweets
<class 'twx.Tweet'> SELECT t1."id", t1."user_id", t1."message", ...
```

Joining tables

Use the *join()* method to *JOIN* additional tables. When a foreign key exists between the source model and the join model, you do not need to specify any additional parameters:

```
>>> my_tweets = Tweet.select().join(User).where(User.username == 'charlie')
```

By default peewee will use an *INNER* join, but you can use *LEFT OUTER*, *RIGHT OUTER*, *FULL*, or *CROSS* joins as well:

```
users = (User
         .select(User, fn.Count(Tweet.id).alias('num_tweets'))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User)
         .order_by(fn.Count(Tweet.id).desc()))
for user in users:
    print user.username, 'has created', user.num_tweets, 'tweet(s).'
```

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, related_name='relationships')
    to_user = ForeignKeyField(User, related_name='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))
```

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).alias('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: By specifying an alias on the join condition, you can control the attribute peewee will assign the joined instance to. In the previous example, we used the following *join*:

```
(User.id == ActivityLog.object_id).alias('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
```

Joining on Multiple Tables

When calling *join()*, peewee will use the *last joined table* as the source table. For example:

```
User.select().join(Tweet).join(Comment)
```

This query will result in a join from *User* to *Tweet*, and another join from *Tweet* to *Comment*.

If you would like to join the same table twice, use the *switch()* method:

```
# Join the Artist table on both `Ablum` and `Genre`.
Artist.select().join(Album).switch(Artist).join(Genre)
```

Implementing Many to Many

Peewee does not provide a *field* for many to many relationships the way that django does – this is because the field really is hiding an intermediary table. To implement many-to-many 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:

```
last = None
for student_course in query:
    student = student_course.student
    if student != last:
        last = student
        print 'Student: %s' % student.name
    print '    - %s' % 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, you can get a big boost by using *ManyToManyField*. Check out the *Fields* extension module for details.

Modeling students and courses using *ManyToManyField*:

```
from peewee import *
from playhouse.fields import ManyToManyField

db = SqliteDatabase('school.db')

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

class Student(BaseModel):
    name = CharField()

class Course(BaseModel):
    name = CharField()
    students = ManyToManyField(Student, related_name='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()
```

For more examples, see:

- *ManyToManyField.add()*
- *ManyToManyField.remove()*
- *ManyToManyField.clear()*

- *ManyToManyField.get_through_model()*

Self-joins

Peewee supports several methods for 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', related_name='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:

```
join_query = Category.select().where(Category.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 t3."id"
  FROM "category" AS t3
  WHERE (t3."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")
```

Performance Techniques

This section outlines some techniques for improving performance when using peewee.

Avoiding N+1 queries

The term *N+1 queries* 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.

Note: 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.
         .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.

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 two approaches to avoiding *O(n)* queries in this situation. We can either:

- 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.
- Fetch both users and tweets in a single query. User data will be duplicated, so peewee will de-dupe it and aggregate the tweets as it iterates through the result set. This method involves a lot of data being transferred over the wire and a lot of logic in Python to de-duplicate rows.

Each solution has its place and, depending on the size and shape of the data you are querying, one may be more performant than the other.

Using prefetch

peewee supports pre-fetching related data using sub-queries. This method requires the use of a special API, *prefetch()*. Pre-fetch, as its name indicates, 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.is_published == True) &
              (Tweet.created_date >= 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_prefetch:
        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.
- In general it is more performant than *aggregate_rows()*.

- Typically a lot less data is transferred over the wire since data is not duplicated.
- There is less Python overhead since we don't have to de-dupe things.
- *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.

Using aggregate_rows

The *aggregate_rows()* approach selects all data in one go and de-dupes things in-memory. Like *prefetch()*, it can work with arbitrarily complex queries. To use this feature We will use a special flag, *aggregate_rows()*, when creating our query. This method tells peewee to de-duplicate any rows that, due to the structure of the JOINs, may be duplicated.

Warning: Because there is a lot of computation involved in de-duping data, it is possible that for some queries *aggregate_rows()* will be **significantly less performant** than using *prefetch()* (described in the previous section) or even issuing *O(n)* simple queries! Profile your code if you're not sure.

```
query = (User
         .select(User, Tweet)  # As in the previous example, we select both tables.
         .join(Tweet, JOIN.LEFT_OUTER)
         .order_by(User.username)  # We need to specify an ordering here.
         .aggregate_rows())  # Tell peewee to de-dupe and aggregate results.

for user in query:
    print user.username
    for tweet in user.tweets:
        print '  ', tweet.message
```

Ordinarily, `user.tweets` would be a *SelectQuery* and iterating over it would trigger an additional query. By using *aggregate_rows()*, though, `user.tweets` is a Python `list` and no additional query occurs.

Note: We used a *LEFT OUTER* join to ensure that users with zero tweets would also be included in the result set.

Below is an example of how we might fetch several users and any tweets they created within the past week. Because we are filtering the tweets and the user may not have any tweets, we need our *WHERE* clause to allow *NULL* tweet IDs.

```
week_ago = datetime.date.today() - datetime.timedelta(days=7)
query = (User
         .select(User, Tweet)
         .join(Tweet, JOIN.LEFT_OUTER)
         .where(
             (Tweet.id >> None) | (
                 (Tweet.is_published == True) &
                 (Tweet.created_date >= week_ago)))
         .order_by(User.username, Tweet.created_date.desc())
         .aggregate_rows())

for user in query:
    print user.username
    for tweet in user.tweets:
        print '  ', tweet.message
```

Some things to consider when using *aggregate_rows()*:

- You must specify an ordering for each table that is joined on so the rows can be aggregated correctly, sort of similar to itertools.groupby.
- Do not mix calls to *aggregate_rows()* with `LIMIT` or `OFFSET` clauses, or with *get()* (which applies a `LIMIT 1` SQL clause). Since the aggregate result set may contain more than one item due to rows being duplicated, limits can lead to incorrect behavior. Imagine you have three users, each of whom has 10 tweets. If you run a query with a `LIMIT 5`, then you will only receive the first user and their first 5 tweets.
- In general the Python overhead of de-duplicating data can make this method less performant than *prefetch()*, and sometimes even less performan than simply issuing *O(n)* simple queries! When in doubt profile.
- Because every column from every table is included in each row tuple returned by the cursor, this approach can use a lot more bandwidth than *prefetch()*.

Iterating over lots of rows

By default peewee will cache the rows returned when iterating of a *SelectQuery*. 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 using the *naive()* method. This method speeds up the construction of peewee model instances from raw cursor data. See the *naive()* documentation for more details on this optimization.

```
for stat in stats.naive().iterator():
    serializer.serialize_object(stat)
```

You can also see performance improvements by using the *dicts()* and *tuples()* methods.

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. To speed up model creation, you can:

- Call *naive()*, which will not construct a graph and simply patch all attributes from the row directly onto a model instance.
- Use *dicts()* or *tuples()*.

Speeding up Bulk Inserts

See the *Bulk inserts* section for details on speeding up bulk insert operations.

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.

Consider this code:

```
db.begin()  # Open a new transaction.
try:
    save_some_objects()
except ErrorSavingData:
    db.rollback()  # Uh-oh! Let's roll-back any partial changes.
    error_saving = True

create_report(error_saving=error_saving)
db.commit()  # What happens here??
```

If the `ErrorSavingData` exception gets raised, we call rollback, but because we are not using the `~Database.atomic` context manager, **no new transaction is begun**. The call to `commit()` will fail because no transaction is active!

On the other hand, consider this:

```
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().
        db.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')

        # 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:')

with db.transaction():
    # 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.
```

Note: 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, databases are initialized with `autocommit=True`, you can turn this on and off at runtime if you like. If you choose to disable autocommit, then you must explicitly call *Database.begin()* to begin a transaction, and commit or roll back.

The behavior below is roughly the same as the context manager and decorator:

```
db.set_autocommit(False)
db.begin()
try:
    user.delete_instance(recursive=True)
except:
    db.rollback()
    raise
else:
    try:
        db.commit()
    except:
        db.rollback()
        raise
finally:
    db.set_autocommit(True)
```

If you would like to manually control *every* transaction, simply turn autocommit off when instantiating your database:

```
db = SqliteDatabase(':memory:', autocommit=False)

db.begin()
User.create(username='somebody')
db.commit()
```

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*
- *SqliteQ*
- *Sqlite User-Defined Functions*
- *apsw, an advanced sqlite driver*
- *BerkeleyDB backend*
- *Sqlcipher backend*
- *Postgresql Extensions*

High-level features

- *Fields*
- *Shortcuts*
- *Hybrid Attributes*
- *Signal support*
- *DataSet*
- *Key/Value Store*
- *Generic foreign keys*
- *CSV Utils*

Database management and framework integration

- *pwiz, a model generator*
- *Schema Migrations*
- *Connection pool*
- *Reflection*
- *Database URL*
- *Read Slaves*
- *Test Utils*
- *pskel*
- *Flask Utils*
- *Django Integration*

Sqlite Extensions

The SQLite extensions module provides support for some interesting sqlite-only features:

- Define custom aggregates, collations and functions.
- Support for FTS3/4 (sqlite full-text search) with *BM25 ranking*.
- C extension providing fast implementations of ranking and other utility functions.
- Support for the new FTS5 search extension.
- Specify isolation level in transactions.
- Support for virtual tables and SQLite C extensions.
- Support for the closure table extension, which allows efficient querying of heirarchical tables.

sqlite_ext API notes

class **SqliteExtDatabase**(*database*, *pragmas=()*, *c_extensions=True*, ***kwargs*)

Parameters

- **pragmas** – A list or tuple of 2-tuples containing PRAGMA settings to configure on a per-connection basis.

- **c_extensions** (*bool*) – Boolean flag indicating whether to use the fast implementations of various SQLite user-defined functions. If Cython was installed when you built `peewee`, then these functions should be available. If not, Peewee will fall back to using the slower pure-Python functions.

Subclass of the *SqliteDatabase* that provides some advanced features only offered by Sqlite.

•Register custom aggregates, collations and functions

•Support for SQLite virtual tables and C extensions

•Specify a row factory

•Advanced transactions (specify isolation level)

aggregate(*name=None*[, *num_params=-1*])
Class-decorator for registering custom aggregation functions.

Parameters

- **name** – string name for the aggregate, defaults to the name of the class.
- **num_params** – integer representing number of parameters the aggregate function accepts. The default value, `-1`, indicates the aggregate can accept any number of parameters.

```
@db.aggregate('product', 1)
class Product(object):
    """Like sum, except calculate the product of a series of numbers."""
    def __init__(self):
        self.product = 1

    def step(self, value):
        self.product *= value

    def finalize(self):
        return self.product

# To use this aggregate:
product = (Score
           .select(fn.product(Score.value))
           .scalar())
```

unregister_aggregate(name):
Unregister the given aggregate function.

collation([*name*])
Function decorator for registering a custom collation.

Parameters **name** – string name to use for this collation.

```
@db.collation()
def collate_reverse(s1, s2):
    return -cmp(s1, s2)

# To use this collation:
Book.select().order_by(collate_reverse.collation(Book.title))
```

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.

unregister_collation(name):
Unregister the given collation function.

func([*name*[, *num_params*]])
Function decorator for registering user-defined functions.

Parameters

- **name** – name to use for this function.
- **num_params** – number of parameters this function accepts. If not provided, peewee will introspect the function for you.

```
@db.func()
def title_case(s):
    return s.title()

# Use in the select clause...
titled_books = Book.select(fn.title_case(Book.title))

@db.func()
def sha1(s):
    return hashlib.sha1(s).hexdigest()

# Use in the where clause...
user = User.select().where(
    (User.username == username) &
    (fn.sha1(User.password) == password_hash)).get()
```

unregister_function(name):
Unregister the given user-defiend function.

load_extension(*extension*)
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')
```

unload_extension(name):
Unload the given SQLite extension.

class VirtualModel
Subclass of *Model* that signifies the model operates using a virtual table provided by a sqlite extension.

Creating a virtual model is easy, simply subclass `VirtualModel` and specify the extension module and any options:

```
class MyVirtualModel(VirtualModel):
    class Meta:
        database = db
        extension_module = 'nextchar'
        extension_options = {}
```

Meta.extension_module = 'name of sqlite extension'

Meta.extension_options = {'tokenize': 'porter', etc}
SQLite virtual tables often support configuration via arbitrary key/value options which are included in the

CREATE TABLE statement. To configure a virtual table, you can specify options like this:

```
class SearchIndex(FTSModel):
    content = SearchField()
    metadata = SearchField()

    class Meta:
        database = my_db
        extension_options = {
            'prefix': [2, 3],
            'tokenize': 'porter',
        }
```

class FTSModel

Model class that provides support for Sqlite's full-text search extension. Models 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 docid field which is automatically created and managed by SQLite (unless you choose to explicitly set it during model creation). Lookups on this column **are performant**.

sqlite_ext provides a *SearchField* field class which should be used on FTSModel implementations instead of the regular peewee field types. This will help prevent you accidentally creating invalid column constraints.

Because of the lack of secondary indexes, it usually makes sense to use the docid primary key as a pointer to a row in a regular table. For example:

```
class Document(Model):
    author = ForeignKeyField(User, related_name='documents')
    title = TextField(null=False, unique=True)
    content = TextField(null=False)
    timestamp = DateTimeField()

    class Meta:
        database = db


class DocumentIndex(FTSModel):
    title = SearchField()
    content = SearchField()

    class Meta:
        database = db
        # Use the porter stemming algorithm to tokenize content.
        extension_options = {'tokenize': 'porter'}
```

To store a document in the document index, we will INSERT a row into the DocumentIndex table, manually setting the docid:

```
def store_document(document):
    DocumentIndex.insert({
        DocumentIndex.docid: 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`:

```
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.docid))
            .where(DocumentIndex.match(phrase))
            .order_by(DocumentIndex.bm25()))
```

Warning: All SQL queries on `FTSModel` classes will be slow **except** full-text searches and `docid` lookups.

Continued examples:

```
# Use the "match" operation for FTS queries.
matching_docs = (DocumentIndex
                 .select()
                 .where(DocumentIndex.match('some query')))

# To sort by best match, use the custom "rank" function.
best = (DocumentIndex
        .select()
        .where(DocumentIndex.match('some query'))
        .order_by(DocumentIndex.rank()))

# Or use the shortcut method:
best = DocumentIndex.search('some phrase')

# Peewee allows you to specify weights for columns.
# Matches in the title will be 2x more valuable than matches
# in the content field:
best = DocumentIndex.search(
    'some phrase',
    weights=[2.0, 1.0],
)
```

Examples using the BM25 ranking algorithm:

```
# you can also use the BM25 algorithm to rank documents:
best = (DocumentIndex
        .select()
        .where(DocumentIndex.match('some query'))
        .order_by(DocumentIndex.bm25()))

# There is a shortcut method for bm25 as well:
best_bm25 = DocumentIndex.search_bm25('some phrase')

# BM25 allows you to specify weights for columns.
```

```
# Matches in the title will be 2x more valuable than matches
# in the content field:
best_bm25 = DocumentIndex.search_bm25(
    'some phrase',
    weights=[2.0, 1.0],
)
```

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 code snippet illustrating how to implement this with peewee:

```
class Blog(Model):
    title = CharField()
    pub_date = DateTimeField()
    content = TextField()  # we want to search this.

    class Meta:
        database = db

class BlogIndex(FTSModel):
    content = SearchField()

    class Meta:
        database = db
        extension_options = {'content': Blog.content}

db.create_tables([Blog, BlogIndex])

# Now, we can manage content in the FTSBlog.  To populate it with
# content:
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. However, content will need to be manually moved to/from the associated `FTSModel`.

FTSModel API methods:

classmethod `create_table` ([*fail_silently=False* [, ***options*]])

Parameters

- **`fail_silently`** (*boolean*) – do not re-create if table already exists.
- **`options`** – options passed along when creating the table, e.g. `content`.

classmethod `match` (*term*)

Shorthand for generating a `MATCH` expression for the given term(s).

```
query = (DocumentIndex
         .select()
         .where(DocumentIndex.match('search phrase')))
```

```
for doc in query:
    print 'match: ', doc.title
```

classmethod search (*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*]]])

Shorthand way of searching for a term and sorting results by the quality of the match. This is equivalent to the *rank()* example code presented below.

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`.

```
# 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 rank ([*col1_weight*, *col2_weight...coln_weight*])

Generate an expression that will calculate and return the quality of the search match. This `rank` can be used to sort the search results. The lower the `rank`, the better the 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.

```
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 search_bm25 (*term*[, *weights=None*[, *with_score=False*[, *score_alias='score'*]]])

Shorthand way of searching for a term and sorting results by the quality of the match, as determined by the BM25 algorithm. This is equivalent to the *bm25()* example code presented below.

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`.

```
# 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 bm25([*col1_weight*, *col2_weight...coln_weight*])

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, and the lower the value the better the match.

The `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.

```
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
```

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 SearchField([*unindexed=False*[, *db_column=None*[, *coerce=None*]]])

Parameters

- **unindexed** – Whether the contents of this field should be excluded from the full-text search index.
- **db_column** – Name of the underlying database column.
- **coerce** – Function used to convert the value from the database into the appropriate Python format.

class JSONField

Field class suitable for working with JSON stored and manipulated using the JSON1 extension.

Most functions that operate on JSON fields take a `path` argument. The JSON documents specify that the path should begin with `'$'` followed by zero or more instances of `'.objectlabel'` or `'[arrayindex]'`. Peewee simplifies this by allowing you to omit the `'$'` character and just specify the path you need or `None` for an empty path:

- `path=''` -> `'$'`
- `path='tags'` -> `'$.tags'`
- `path='[0][1].bar'` -> `'$[0][1].bar'`
- `path='metadata[0]'` -> `'$.metadata[0]'`
- `path='user.data.email'` -> `'$.user.data.email'`

`length`(*path=None*)

Return the number of items in a JSON array at the given path. If the path is omitted, then return the number of items in the top-level array.

SQLite documentation.

`extract`(*path*)

Return the value at the given path. If the value is a JSON object or array, it will be decoded into a `dict` or `list`. If the value is a scalar type, string or `null` then it will be returned as the appropriate Python type.

SQLite documentation.

Example:

```
# data looks like {'post': {'title': 'post 1', 'body': '...'}, ...}
query = (Post
         .select(Post.data.json_extract('post.title'))
         .tuples())

# Only the `title` value is extracted from the JSON data.
for title, in query:
    print title
```

`set`(*path*, *value*, *path2*, *value2...*)

Set values stored in the input JSON string using the given path/value pairs. The `set` function returns a **new** JSON string formed by updating the input JSON with the given path/value pairs.

If the path does not exist, it **will** be created.

Similarly, if the path does exist, it **will** be overwritten.

SQLite documentation. Example:

```
PostAlias = Post.alias()
set_query = (PostAlias
             .select(PostAlias.data.set(
                 'title', 'New title',
                 'tags', ['list', 'of', 'new', 'tags'],
                 'totally.new.field', 3,
                 'status.published', True))
             .where(PostAlias.id == Post.id))

# Update multiple fields at one time on the Post
# with the title "Old title".
query = (Post
         .update(data=set_query)
         .where(Post.data.extract('title') == 'Old title'))
```

```
query.execute()

post = (Post
        .select()
        .where(Post.data.extract('title') == 'New title')
        .get())

# Our new data has been added, even nested objects that did not
# exist before. Any pre-existing data has also been preserved,
# provided it was not over-written.
assert post.data == {
    'title': 'New title',
    'tags': ['list', 'of', 'new', 'tags'],
    'totally': {'new': {'field: 3}},
    'status': {'published': True, 'draft': False},
    'other-field': ['this', 'was', 'here', 'before'],
    'another-old-field': 'etc, etc'}
```

insert(*path*, *value*[, *path2*, *value2*...])

Insert the given path/value pairs into the JSON string stored in the field. The `insert` function returns a **new** JSON string formed by updating the input JSON with the given path/value pairs.

If the path already exists, it will **not** be overwritten.

SQLite documentation.

replace(*path*, *value*[, *path2*, *value2*...])

Replace values stored in the input JSON string using the given path/value pairs. The `replace` function returns a **new** JSON string formed by updating the input JSON with the given path/value pairs.

If the path does not exist, it will **not** be created.

SQLite documentation.

remove(**paths*)

Remove values referenced by the given path(s). The `remove` function returns a **new** JSON string formed by removing the specified paths from the input JSON string.

The process for removing fields from a JSON column is similar to the way you *`set()`* them. For a code example, see *updating JSON data*.

SQLite documentation.

json_type([*path=None*])

Return a string indicating the type of object stored in the field. You can optionally supply a path to specify a sub-item. The types of objects are:

- 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

SQLite documentation.

children(*[path=None]*)

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.

For examples, see my blog post on JSON1.

SQLite documentation.

tree(*[path=None]*)

The `tree` function corresponds to `json_tree`, a table-valued function that walks the JSON value provided and recursively returns all descendants of the given root node. If a path is specified, then that path is treated as the root node element.

The rows returned by calls to `tree()` have the same attributes as rows returned by calls to *children()*.

For examples, see my blog post on JSON1.

SQLite documentation.

class PrimaryKeyAutoIncrementField

Subclass of *PrimaryKeyField* that uses a monotonically-increasing value for the primary key. This differs from the default SQLite primary key, which simply uses the "max + 1" approach to determining the next ID.

class RowIDField

Subclass of *PrimaryKeyField* that provides access to the underlying `rowid` field used internally by SQLite.

Note: When added to a Model, this field will act as the primary key. However, this field will not be included by default when selecting rows from the table.

---

class DocIDField

Subclass of *PrimaryKeyField* that provides access to the underlying `docid` field used internally by SQLite's FTS3/4 virtual tables.

Note: This field should not be created manually, as it is only needed on `FTSModel` classes, which include it already.

---

match(*lhs*, *rhs*)

Generate a SQLite *MATCH* expression for use in full-text searches.

```
Document.select().where(match(Document.content, 'search term'))
```

class FTS5Model

Model class that should be used to implement virtual tables using the FTS5 extension. Documentation on the FTS5 extension can be found here. This extension behaves very similarly to the FTS3 and FTS4 extensions, and the `FTS5Model` supports many of the same APIs as *FTSModel*.

The `FTS5` extension is more strict in enforcing that no column define any type or constraints. For this reason, only *SearchField* objects can be used with `FTS5Model` implementations.

Additionally, `FTS5` 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'*]]])

Shorthand way of searching for a term and sorting results by the quality of the match. This is equivalent to the built-in `rank` value provided by the `FTS5` extension.

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`.

```
# 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, the `search_bm25` method is the same as the *FTS5Model.search()* method.

classmethod VocabModel([*table_type='row'|'col'*[, *table_name=None*]])

Parameters

- **table_type** – Either `'row'` or `'col'`.
- **table_name** – Name for the vocab table. If not specified, will be "fts5tablename_v".

ClosureTable(*model_class*[, *foreign_key=None*[, *referencing_class=None*, *referencing_key=None*]])

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 heirarchical 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 heirarchical 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, related_name='_known_by')

    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')
```

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** – The 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.

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 code:

```
db = SqliteExtDatabase('my_database.db')
db.load_extension('/path/to/closure')

class Category(Model):
    name = CharField()
    parent = ForiegnKeyField('self', index=True, null=True)  # Required.

    class Meta:
        database = db

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
# Desktops
# Hard-drives
# Monitors
# LCD Monitors
# LED Monitors
```

The `VirtualTable` returned by this function contains a handful of interesting methods. The model will be a subclass of *BaseClosureTable*.

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.

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.

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 install the vtfunc module. The vtfunc module is available on GitHub or can be installed using pip.

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() (f)
- DATE * strip_tz() (f) * human_delta() (f) * mintdiff() (a) * avgtdiff() (a) * duration() (a) * date_series() (t)
- FILE * file_ext() (f) * file_read() (f)
- HELPER * gzip() (f) * gunzip() (f) * hostname() (f) * toggle() (f) * setting() (f) * clear_toggles() (f) * clear_settings() (f)
- MATH * randomrange() (f) * gauss_distribution() (f) * sqrt() (f) * tonumber() (f) * mode() (a) * minrange() (a) * avgrange() (a) * range() (a) * median() (a) (requires cython)
- STRING * substr_count() (f) * strip_chars() (f) * md5() (f) * sha1() (f) * sha256() (f) * sha512() (f) * adler32() (f) * crc32() (f) * damerau_levenshtein_dist() (f) (requires cython) * levenshtein_dist() (f) (requires cython) * str_dist() (f) (requires cython) * regex_search() (t)

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:
        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`.

BerkeleyDB backend

BerkeleyDB provides a SQLite-compatible API. BerkeleyDB's SQL API has many advantages over SQLite:

- Higher transactions-per-second in multi-threaded environments.
- Built-in replication and hot backup.
- Fewer system calls, less resource utilization.
- Multi-version concurrency control.

For more details, Oracle has published a short technical overview.

In order to use peewee with BerkeleyDB, you need to compile BerkeleyDB with the SQL API enabled. Then compile the Python SQLite driver against BerkeleyDB's sqlite replacement.

Begin by downloading and compiling BerkeleyDB:

```
wget http://download.oracle.com/berkeley-db/db-6.0.30.tar.gz
tar xzf db-6.0.30.tar.gz
cd db-6.0.30/build_unix
export CFLAGS='-DSQLITE_ENABLE_FTS3=1 -DSQLITE_ENABLE_FTS3_PARENTHESIS=1 -DSQLITE_
↪ENABLE_UPDATE_DELETE_LIMIT -DSQLITE_SECURE_DELETE -DSQLITE_SOUNDEX -DSQLITE_ENABLE_
↪RTREE=1 -fPIC'
../dist/configure --enable-static --enable-shared --enable-sql --enable-sql-compat
make
sudo make prefix=/usr/local/ install
```

Then get a copy of the standard library SQLite driver and build it against BerkeleyDB:

```
git clone https://github.com/ghaering/pysqlite
cd pysqlite
sed -i "s|#||g" setup.cfg
python setup.py build
sudo python setup.py install
```

You can also find up-to-date step by step instructions on my blog.

class **BerkeleyDatabase**(*database*, ***kwargs*)

Parameters

- **multiversion** (*bool*) – Enable multiversion concurrency control. Default is `False`.
- **page_size** (*int*) – Set the page size `PRAGMA`. This option only works on new databases.
- **cache_size** (*int*) – Set the cache size `PRAGMA`.

Subclass of the *SqliteExtDatabase* that supports connecting to BerkeleyDB-backed version of SQLite.

classmethod check_pysqlite()
Check whether `pysqlite2` was compiled against the BerkeleyDB SQLite. Returns `True` or `False`.

classmethod check_libsqlite()
Check whether `libsqlite3` is the BerkeleyDB SQLite implementation. Returns `True` or `False`.

Sqlcipher backend

- Although this extention's code is short, it has not been properly peer-reviewed yet and may have introduced vulnerabilities.
- The code contains minimum values for *passphrase* length and *kdf_iter*, as well as a default value for the later. **Do not** regard these numbers as advice. Consult the docs at http://sqlcipher.net/sqlcipher-api/ and security experts.

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*, *kdf_iter=64000*, ***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 (or an error is raised), but it is *strongly advised* to enforce better passphrase strength criteria in your implementation.
- **kdf_iter** – [Optional] number of PBKDF2 iterations.

•If the `database` file doesn't exist, it will be *created* with encryption by a key derived from `passhprase` with `kdf_iter` PBKDF2 iterations.

•When trying to open an existing database, `passhprase` and `kdf_iter` should be *identical* to the ones used when it was created.

Notes:

- [Hopefully] there's no way to tell whether the passphrase is wrong or the file is corrupt. In both cases – *the first time we try to acces the database* – a `DatabaseError` error is raised, with the *exact* message: `"file is encrypted or is not a database"`.

  As mentioned above, this only happens when you *access* the databse, so if you need to know *right away* whether the passphrase was correct, you can trigger this check by calling [e.g.] *get_tables()* (see example below).
- Most applications can expect failed attempts to open the database (common case: prompting the user for `passphrase`), so the database can't be hardwired into the `Meta` of model classes. To defer initialization, pass *None* in to the database.

Example:

```
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 = CharField(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:
        # We only allow a specific [somewhat cryptic] error message.
        if exc.args[0] != 'file is encrypted or is not a database':
            raise exc
        else:
            tell_user_the_passphrase_was_wrong()
            db.init(None)  # Reset the db.
    else:
        # The password was correct.
        right_passphrase = True
```

See also: a slightly more elaborate example.

Postgresql Extensions

The postgresql extensions module provides a number of "postgres-only" functions, currently:

- *hstore support*
- *json support*, including `jsonb` for Postgres 9.4.
- *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
```

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.

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 keys or partial dictionary:

```
>>> f = House.features
>>> House.select().where(f.contains('garage')) # <-- all houses w/garage key
>>> House.select().where(f.contains(['garage', 'bath'])) # <-- all houses w/garage &␣
↪bath
>>> House.select().where(f.contains({'garage': '2 cars'})) # <-- houses w/2-car garage
```

Suppose you want to do an atomic update to the house:

```
>>> f = House.features
>>> new_features = House.features.update({'bath': '2.5 bath', 'sqft': '1100'})
>>> query = House.update(features=new_features)
>>> query.where(House.id == h.id).execute()
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=f.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=f.delete('garage', 'sqft'))
```

You can select just keys, just values, or zip the two:

```
>>> f = House.features
>>> for h in House.select(House.address, f.keys().alias('keys')):
...     print h.address, h.keys

123 Main St [u'bath', u'garage']

>>> for h in House.select(House.address, f.values().alias('vals')):
...     print h.address, h.vals

123 Main St [u'2 bath', u'2 cars']

>>> for h in House.select(House.address, f.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:

```
>>> f = House.features
>>> for h in House.select(House.address, f.slice('garage').alias('garage_data')):
...     print h.address, h.garage_data

123 Main St {'garage': '2 cars'}
```

You can check for the existence of a key and filter rows accordingly:

```
>>> for h in House.select(House.address, f.exists('garage').alias('has_garage')):
...     print h.address, h.has_garage

123 Main St True

>>> for h in House.select().where(f.exists('garage')):
...     print h.address, h.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))
```

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')  # note

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://wtf.charlesleifer.com/api/offense/')
booking = APIResponse.request('http://wtf.charlesleifer.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.

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(query):
    return Blog.select().where(
        (Blog.status == Blog.STATUS_PUBLISHED) &
        Match(Blog.content, query))
```

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()
```

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))
```

Note: If you are using the *TSVectorField*, it will automatically be created with a GIN index.

postgres_ext API notes

class **PostgresqlExtDatabase**(*database*[, *server_side_cursors=False*[, *register_hstore=True*[, ...]]])

Identical to *PostgresqlDatabase* but required in order to support:

- *Server-side cursors*
- *ArrayField*
- *DateTimeTZField*
- *JSONField*
- *BinaryJSONField*
- *HStoreField*
- *TSVectorField*

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.

If using `server_side_cursors`, also be sure to wrap your queries with *ServerSide()*.

If you do not wish to use the HStore extension, you can specify `register_hstore=False`.

Warning: The *PostgresqlExtDatabase* by default will attempt to register the `HSTORE` extension. Most distributions and recent versions include this, but in some cases the extension may not be available. If you **do not** plan to use the *HStore features of peewee*, you can pass `register_hstore=False` when initializing your *PostgresqlExtDatabase*.

ServerSide(*select_query*)

Wrap the given select query in a transaction, and call it's *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).

Parameters **select_query** – a *SelectQuery* instance.

Return type `generator`

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*[, *dimensions=1*]])

Field capable of storing arrays of the provided *field_class*.

Parameters

- **field_class** – a subclass of `Field`, e.g. *IntegerField*.
- **dimensions** (*int*) – dimensions of array.

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*.

keys()

Returns the keys for a given row.

```
>>> f = House.features
>>> for h in House.select(House.address, f.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, f.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, f.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.

```
>>> f = House.features
>>> for h in House.select(House.address, f.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, f.exists('garage').
↪alias('has_garage')):
...     print h.address, h.has_garage

123 Main St True

>>> for h in House.select().where(f.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.

```
>>> f = House.features
>>> House.select().where(f.contains('garage')) # <-- all houses w/garage key
>>> House.select().where(f.contains(['garage', 'bath'])) # <-- all houses w/
↪garage & bath
>>> House.select().where(f.contains({'garage': '2 cars'})) # <-- houses w/2-
↪car garage
```

contains_any(**keys*)

Parameters **keys** – One or more keys to search for.

Query rows for the existince of *any* key.

class JSONField(*dumps=None*, **args*, ***kwargs*)

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.

Parameters **dumps** – The default is to call json.dumps() or the dumps function. You can override this method to create a customized JSON wrapper.

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()

    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.
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*)

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.

Parameters **dumps** – The default is to call json.dumps() or the dumps function. You can override this method to create a customized JSON wrapper.

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))
```

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(query):
    return Blog.select().where(
        (Blog.status == Blog.STATUS_PUBLISHED) &
        Match(Blog.content, query))
```

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.

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()`.
```

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 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*)

The *DataSet* class provides a high-level API for working with relational databases.

Parameters **url** (*str*) – A database URL. See *Database URL* for examples.

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*)
The *Table* class 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')

# 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.

Django Integration

The Django ORM provides a very high-level abstraction over SQL and as a consequence is in some ways limited in terms of flexibility or expressiveness. I wrote a blog post describing my search for a "missing link" between Django's ORM and the SQL it generates, concluding that no such layer exists. The `djpeewee` module attempts to provide an easy-to-use, structured layer for generating SQL queries for use with Django's ORM.

A couple use-cases might be:

- Joining on fields that are not related by foreign key (for example UUID fields).
- Performing aggregate queries on calculated values.
- Features that Django does not support such as `CASE` statements.
- Utilizing SQL functions that Django does not support, such as `SUBSTR`.
- Replacing nearly-identical SQL queries with reusable, composable data-structures.

Below is an example of how you might use this:

```
# Django model.
class Event(models.Model):
    start_time = models.DateTimeField()
    end_time = models.DateTimeField()
    title = models.CharField(max_length=255)

# Suppose we want to find all events that are longer than an hour.  Django
# does not support this, but we can use peewee.
from playhouse.djpeewee import translate
P = translate(Event)
query = (P.Event
         .select()
         .where(
             (P.Event.end_time - P.Event.start_time) > timedelta(hours=1)))

# Now feed our peewee query into Django's `raw()` method:
sql, params = query.sql()
Event.objects.raw(sql, params)
```

Foreign keys and Many-to-many relationships

The *translate()* function will recursively traverse the graph of models and return a dictionary populated with everything it finds. Back-references are not searched by default, but can be included by specifying `backrefs=True`.

Example:

```
>>> from django.contrib.auth.models import User, Group
>>> from playhouse.djpeewee import translate
>>> translate(User, Group)
{'ContentType': peewee.ContentType,
 'Group': peewee.Group,
 'Group_permissions': peewee.Group_permissions,
 'Permission': peewee.Permission,
 'User': peewee.User,
 'User_groups': peewee.User_groups,
 'User_user_permissions': peewee.User_user_permissions}
```

As you can see in the example above, although only *User* and *Group* were passed in to `translate()`, several other models which are related by foreign key were also created. Additionally, the many-to-many "through" tables were created as separate models since peewee does not abstract away these types of relationships.

Using the above models it is possible to construct joins. The following example will get all users who belong to a group that starts with the letter "A":

```
>>> P = translate(User, Group)
>>> query = P.User.select().join(P.User_groups).join(P.Group).where(
...     fn.Lower(fn.Substr(P.Group.name, 1, 1)) == 'a')
>>> sql, params = query.sql()
>>> print sql  # formatted for legibility
SELECT t1."id", t1."password", ...
FROM "auth_user" AS t1
INNER JOIN "auth_user_groups" AS t2 ON (t1."id" = t2."user_id")
INNER JOIN "auth_group" AS t3 ON (t2."group_id" = t3."id")
WHERE (Lower(Substr(t3."name", %s, %s)) = %s)
```

djpeewee API

translate(**models*, ***options*)

Translate the given Django models into roughly equivalent peewee models suitable for use constructing queries. Foreign keys and many-to-many relationships will be followed and models generated, although back references are not traversed.

Parameters

- **models** – One or more Django model classes.
- **options** – A dictionary of options, see note below.

Returns A dict-like object containing the generated models, but which supports dotted-name style lookups.

The following are valid options:

- `recurse`: Follow foreign keys and many to many (default: `True`).
- `max_depth`: Maximum depth to recurse (default: `None`, unlimited).
- `backrefs`: Follow backrefs (default: `False`).
- `exclude`: A list of models to exclude.

Fields

This module also contains several field classes that implement additional logic like encryption and compression. There is also a *ManyToManyField* that makes it easy to work with simple many-to-many relationships.

These fields can be found in the `playhouse.fields` module.

class ManyToManyField(*rel_model*[, *related_name=None*[, *through_model=None*]])

Parameters

- **rel_model** – *Model* class.
- **related_name** (*str*) – Name for the automatically-created backref. If not provided, the pluralized version of the model will be used.
- **through_model** – *Model* to use for the intermediary table. If not provided, a simple through table will be automatically created.

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:

```
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, related_name='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 *SelectQuery* 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

>>> 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.

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 first.

Associate `value` with the current instance. You can pass in a single model instance, a list of model instances, or even a *SelectQuery*.

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 *SelectQuery*.

Disassociate `value` from the current instance. Like *add()*, you can pass in a model instance, a list of model instances, or even a *SelectQuery*.

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
In some instances, you may need to obtain a reference to a through model before that model is actually defined. In order to avoid weird circular logic, you can use the `DeferredThroughModel` as a placeholder, then "fill it in" when you're ready.

Example:

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

NoteThroughDeferred = DeferredThroughModel()  # Create placeholder.

class Note(Model):
    text = TextField()
    users = ManyToManyField(User, through_model=NoteThroughDeferred)

class NoteThrough(Model):
    user = ForeignKeyField(User)
    note = ForeignKeyField(Note)
    sort_order = IntegerField(default=0)

# Now that all the models are defined, we can replace the placeholder
# with the actual through model implementation.
NoteThroughDeferred.set_model(NoteThrough)
```

set_model(*model_class*)
Initialize the deferred placeholder with the appropriate model class.

class CompressedField([*compression_level=6*[, *algorithm='zlib'*[, ***kwargs*]]])
`CompressedField` stores compressed data using the specified algorithm. This field extends *BlobField*, transparently storing a compressed representation of the data in the database.

Parameters

- **compression_level** (*int*) – A value from 0 to 9.
- **algorithm** (*str*) – Either `'zlib'` or `'bz2'`.

class PasswordField([*iterations=12*[, ***kwargs*]])
`PasswordField` stores a password hash and lets you verify it. The password is hashed when it is saved to the database and after reading it from the database you can call `check_password (password) -> bool` on it.

Parameters **iterations** (*int*) – Indicates the work factor, it does 2^n iterations.

Note: This field requires bcrypt, which can be installed by running `pip install bcrypt`.

class **PickledField**([**kwargs*])
A field capable of storing arbitrary Python objects.

Note: If the `cPickle` module is available, it will be used.

Generic foreign keys

The `gfk` module provides a Generic ForeignKey (GFK), similar to Django. A GFK is composed of two columns: an object ID and an object type identifier. The object types are collected in a global registry (`all_models`).

How a *GFKField* is resolved:

1. Look up the object type in the global registry (returns a model class)
2. Look up the model instance by object ID

Note: In order to use Generic ForeignKeys, your application's models *must* subclass `playhouse.gfk.Model`. This ensures that the model class will be added to the global registry.

Note: GFKs themselves are not actually a field and will not add a column to your table.

Like regular ForeignKeys, GFKs support a "back-reference" via the *ReverseGFK* descriptor.

How to use GFKs

1. Be sure your model subclasses `playhouse.gfk.Model`
2. Add a *CharField* to store the `object_type`
3. Add a field to store the `object_id` (usually a *IntegerField*)
4. Add a *GFKField* and instantiate it with the names of the `object_type` and `object_id` fields.
5. (optional) On any other models, add a *ReverseGFK* descriptor

Example:

```
from playhouse.gfk import *

class Tag(Model):
    tag = CharField()
    object_type = CharField(null=True)
    object_id = IntegerField(null=True)
    object = GFKField('object_type', 'object_id')

class Blog(Model):
    tags = ReverseGFK(Tag, 'object_type', 'object_id')
```

```
class Photo(Model):
    tags = ReverseGFK(Tag, 'object_type', 'object_id')
```

How you use these is pretty straightforward hopefully:

```
>>> b = Blog.create(name='awesome post')
>>> Tag.create(tag='awesome', object=b)
>>> b2 = Blog.create(name='whiny post')
>>> Tag.create(tag='whiny', object=b2)

>>> b.tags # <-- a select query
<class '__main__.Tag'> SELECT t1."id", t1."tag", t1."object_type", t1."object_id"␣
↪FROM "tag" AS t1 WHERE ((t1."object_type" = ?) AND (t1."object_id" = ?)) [u'blog',␣
↪1]

>>> [x.tag for x in b.tags]
[u'awesome']

>>> [x.tag for x in b2.tags]
[u'whiny']

>>> p = Photo.create(name='picture of cat')
>>> Tag.create(object=p, tag='kitties')
>>> Tag.create(object=p, tag='cats')

>>> [x.tag for x in p.tags]
[u'kitties', u'cats']

>>> [x.tag for x in Blog.tags]
[u'awesome', u'whiny']

>>> t = Tag.get(Tag.tag == 'awesome')
>>> t.object
<__main__.Blog at 0x268f450>

>>> t.object.name
u'awesome post'
```

GFK API

class **GFKField**([*model_type_field='object_type'*[, *model_id_field='object_id'*]])

Provide a clean API for storing "generic" foreign keys. Generic foreign keys are comprised of an object type, which maps to a model class, and an object id, which maps to the primary key of the related model class.

Setting the GFKField on a model will automatically populate the `model_type_field` and `model_id_field`. Similarly, getting the GFKField on a model instance will "resolve" the two fields, first looking up the model class, then looking up the instance by ID.

class **ReverseGFK**(*model*[, *model_type_field='object_type'*[, *model_id_field='object_id'*]])

Back-reference support for *GFKField*.

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):
        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 `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

    @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)
)
```

Key/Value Store

Provides a simple key/value store, using a dictionary API. By default the the *KeyStore* will use an in-memory sqlite database, but any database will work.

To start using the key-store, create an instance and pass it a field to use for the values.

```
>>> kv = KeyStore(TextField())
>>> kv['a'] = 'A'
>>> kv['a']
'A'
```

Note: To store arbitrary python objects, use the *PickledKeyStore*, which stores values in a pickled *BlobField*.

If your objects are JSON-serializable, you can also use the *JSONKeyStore*, which stores the values as JSON-encoded strings.

Using the *KeyStore* it is possible to use "expressions" to retrieve values from the dictionary. For instance, imagine you want to get all keys which contain a certain substring:

```
>>> keys_matching_substr = kv[kv.key % '%substr%']
>>> keys_start_with_a = kv[fn.Lower(fn.Substr(kv.key, 1, 1)) == 'a']
```

KeyStore API

class **KeyStore**(*value_field*[, *ordered=False*[, *database=None*]])

Lightweight dictionary interface to a model containing a key and value. Implements common dictionary methods, such as `__getitem__`, `__setitem__`, `get`, `pop`, `items`, `keys`, and `values`.

Parameters

- **value_field** (*Field*) – Field instance to use as value field, e.g. an instance of *TextField*.
- **ordered** (*boolean*) – Whether the keys should be returned in sorted order
- **database** (Database) – *Database* class to use for the storage backend. If none is supplied, an in-memory Sqlite DB will be used.

Example:

```
>>> from playhouse.kv import KeyStore
>>> kv = KeyStore(TextField())
>>> kv['a'] = 'foo'
>>> for k, v in kv:
...     print k, v
a foo

>>> 'a' in kv
True
>>> 'b' in kv
False
```

class **JSONKeyStore**([*ordered=False*[, *database=None*]])

Identical to the *KeyStore* except the values are stored as JSON-encoded strings, so you can store complex data-types like dictionaries and lists.

Example:

```
>>> from playhouse.kv import JSONKeyStore
>>> jkv = JSONKeyStore()
>>> jkv['a'] = 'A'
>>> jkv['b'] = [1, 2, 3]
>>> list(jkv.items())
[(u'a', 'A'), (u'b', [1, 2, 3])]
```

class **PickledKeyStore**([*ordered=False*[, *database=None*]])

Identical to the *KeyStore* except *anything* can be stored as a value in the dictionary. The storage for the value will be a pickled *BlobField*.

Example:

```
>>> from playhouse.kv import PickledKeyStore
>>> pkv = PickledKeyStore()
>>> pkv['a'] = 'A'
>>> pkv['b'] = 1.0
>>> list(pkv.items())
[(u'a', 'A'), (u'b', 1.0)]
```

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.

case (*predicate*, *expression_tuples*, *default=None*)

Parameters

- **predicate** – A SQL expression or can be `None`.
- **expression_tuples** – An iterable containing one or more 2-tuples comprised of an expression and return value.
- **default** – default if none of the cases match.

Example SQL case statements:

```
-- case with predicate --
SELECT "username",
  CASE "user_id"
    WHEN 1 THEN "one"
    WHEN 2 THEN "two"
    ELSE "?"
  END
FROM "users";

-- case with no predicate (inline expressions) --
SELECT "username",
  CASE
    WHEN "user_id" = 1 THEN "one"
    WHEN "user_id" = 2 THEN "two"
    ELSE "?"
  END
FROM "users";
```

Equivalent function invocations:

```
User.select(User.username, case(User.user_id, (
  (1, "one"),
  (2, "two")), "?"))

User.select(User.username, case(None, (
  (User.user_id == 1, "one"),  # note the double equals
  (User.user_id == 2, "two")), "?"))
```

You can specify a value for the CASE expression using the `alias()` method:

```
User.select(User.username, case(User.user_id, (
  (1, "one"),
  (2, "two")), "?").alias("id_string"))
```

cast (*node*, *as_type*)

Parameters

- **node** – A peewee *Node*, for instance a `Field` or an `Expression`.
- **as_type** (*str*) – The type name to cast to, e.g. `'int'`.

Returns a function call to cast the node as the given type.

Example:

```
# Find all data points whose numbers are palindromes. We do this by
# casting the number to string, reversing it, then casting the reversed
# string back to an integer.
reverse_val = cast(fn.REVERSE(cast(DataPoint.value, 'str')), 'int')

query = (DataPoint
         .select()
         .where(DataPoint.value == reverse_val))
```

model_to_dict(*model*[, *recurse=True*[, *backrefs=False*[, *only=None*[, *exclude=None*[, *extra_attrs=None*[, *fields_from_query=None*]]]]]])

Convert a model instance (and optionally any related instances) to a dictionary.

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** (SelectQuery) – The *SelectQuery* that created this model instance. Only the fields and values explicitly selected by the query will be serialized.

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': {
    'id': 1,
    'username': 'charlie'
  }
```

```
}

>>> model_to_dict(t2, recurse=False)
{'id': 1, 'message': 'tweet-2', 'user': 1}
```

dict_to_model(*model_class*, *data*[, *ignore_unknown=False*])

Convert a dictionary of data to a model instance, creating related instances where appropriate.

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.

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'
```

class RetryOperationalError

When mixed-in with a vendor-specific *Database* subclass, this class overrides the *execute_sql()* method to automatically reconnect and retry queries that fail due to an OperationalError. The query that failed will be retried only once, and if it fails twice an exception will be raised.

Usage:

```
from peewee import *
from playhouse.shortcuts import RetryOperationalError


class MyRetryDB(RetryOperationalError, MySQLDatabase):
    pass


db = MyRetryDB('my_app')
```

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

post_init Called after a model class has been instantiated and the fields have been populated, for example when being selected as part of a database query.

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*]])

Add the receiver to the internal list of receivers, which will be called whenever the signal is sent.

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

```
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*]])

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.

Parameters

- **`receiver`** (*callable*) – the callback to disconnect
- **`name`** (*string*) – a short alias

```
post_save.disconnect(name='project.cache_buster')
```

`send`(*instance*, **args*, ***kwargs*)

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.

Parameters **`instance`** – a model instance

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()]
```

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 secret
-s	postgres schema	-s public

The following are valid parameters for the engine:

- sqlite
- mysql
- postgresql

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 transaction block, e.g.

```
with my_db.transaction():
    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'),
)
```

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'))
```

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.

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*])

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.

drop_index(*table*, *index_name*)
:param str table Name of the table containing the index to be dropped. :param str index_name: Name of the index to be dropped.

class PostgresqlMigrator(*database*)
Generate migrations for Postgresql databases.

class SqliteMigrator(*database*)
Generate migrations for SQLite databases.

class MySQLMigrator(*database*)
Generate migrations for MySQL databases.

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*.

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*])
Creates an *Introspector* instance suitable for use with the given database.

Parameters

- **database** – a *Database* instance.
- **schema** (*str*) – an optional schema (supported by some databases).

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()
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.

Returns A dictionary mapping table-names to model classes.

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')
```

CSV Utils

This module contains helpers for dumping queries into CSV, and for loading CSV data into a database. CSV files can be introspected to generate an appropriate model class for working with the data. This makes it really easy to explore the data in a CSV file using Peewee and SQL.

Here is how you would load a CSV file into an in-memory SQLite database. The call to *load_csv()* returns a *Model* instance suitable for working with the CSV data:

```
from peewee import *
from playhouse.csv_loader import load_csv
db = SqliteDatabase(':memory:')
ZipToTZ = load_csv(db, 'zip_to_tz.csv')
```

Now we can run queries using the new model.

```
# Get the timezone for a zipcode.
>>> ZipToTZ.get(ZipToTZ.zip == 66047).timezone
'US/Central'

# Get all the zipcodes for my town.
>>> [row.zip for row in ZipToTZ.select().where(
...     (ZipToTZ.city == 'Lawrence') && (ZipToTZ.state == 'KS'))]
[66044, 66045, 66046, 66047, 66049]
```

For more information and examples check out this blog post.

CSV Loader API

load_csv(*db_or_model*, *filename*[, *fields=None*[, *field_names=None*[, *has_header=True*[, *sample_size=10*[, *converter=None*[, *db_table=None*[, ***reader_kwargs*]]]]]]])

Load a CSV file into the provided database or model class, returning a *Model* suitable for working with the CSV data.

Parameters

- **db_or_model** – Either a *Database* instance or a *Model* class. If a model is not provided, one will be automatically generated for you.
- **filename** (*str*) – Path of CSV file to load.
- **fields** (*list*) – A list of `Field` instances mapping to each column in the CSV. This allows you to manually specify the column types. If not provided, and a model is not provided, the field types will be determined automatically.
- **field_names** (*list*) – A list of strings to use as field names for each column in the CSV. If not provided, and a model is not provided, the field names will be determined by looking at the header row of the file. If no header exists, then the fields will be given generic names.

- **has_header** (*bool*) – Whether the first row is a header.
- **sample_size** (*int*) – Number of rows to look at when introspecting data types. If set to 0, then a generic field type will be used for all fields.
- **converter** (*RowConverter*) – a RowConverter instance to use for introspecting the CSV. If not provided, one will be created.
- **db_table** (*str*) – The name of the database table to load data into. If this value is not provided, it will be determined using the filename of the CSV file. If a model is provided, this value is ignored.
- **reader_kwargs** – Arbitrary keyword arguments to pass to the csv.reader object, such as the dialect, separator, etc.

Return type A *Model* suitable for querying the CSV data.

Basic example – field names and types will be introspected:

```
from peewee import *
from playhouse.csv_loader import *
db = SqliteDatabase(':memory:')
User = load_csv(db, 'users.csv')
```

Using a pre-defined model:

```
class ZipToTZ(Model):
    zip = IntegerField()
    timezone = CharField()

load_csv(ZipToTZ, 'zip_to_tz.csv')
```

Specifying fields:

```
fields = [DecimalField(), IntegerField(), IntegerField(), DateField()]
field_names = ['amount', 'from_acct', 'to_acct', 'timestamp']
Payments = load_csv(db, 'payments.csv', fields=fields, field_names=field_names,␣
↪has_header=False)
```

Dumping CSV

dump_csv(*query*, *file_or_name*[, *include_header=True*[, *close_file=True*[, *append=True*[, *csv_writer=None*]]]])

Parameters

- **query** – A peewee *SelectQuery* to dump as CSV.
- **file_or_name** – Either a filename or a file-like object.
- **include_header** – Whether to generate a CSV header row consisting of the names of the selected columns.
- **close_file** – Whether the file should be closed after writing the query data.
- **append** – Whether new data should be appended to the end of the file.
- **csv_writer** – A python csv.writer instance to use.

Example usage:

```
with open('account-export.csv', 'w') as fh:
    query = Account.select().order_by(Account.id)
    dump_csv(query, fh)
```

Connection pool

The `pool` module contains a number of *Database* classes that provide connection pooling for PostgreSQL and MySQL 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 it's 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*]]]])

Mixin class intended to be used with a subclass of *Database*.

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 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.

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.

_connect(**args*, ***kwargs*)
Request a connection from the pool. If there are no available connections a new one will be opened.

_close(*conn*[, *close_conn=False*])
By default *conn* will not be closed and instead will be returned to the pool of available connections. If *close_conn=True*, then *conn* will be closed and *not* be returned to the pool.

manual_close()
Close the currently-open connection without returning it to the pool.

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*.

Read Slaves

The `read_slave` module contains a *Model* subclass that can be used to automatically execute `SELECT` queries against different database(s). This might be useful if you have your databases in a master / slave configuration.

class ReadSlaveModel
Model subclass that will route `SELECT` queries to a different database.

Master and read-slaves are specified using `Model.Meta`:

```
# Declare a master and two read-replicas.
master = PostgresqlDatabase('master')
replica_1 = PostgresqlDatabase('replica_1')
replica_2 = PostgresqlDatabase('replica_2')

# Declare a BaseModel, the normal best-practice.
class BaseModel(ReadSlaveModel):
    class Meta:
        database = master
        read_slaves = (replica_1, replica_2)
```

```
# Declare your models.
class User(BaseModel):
    username = CharField()
```

When you execute writes (or deletes), they will be executed against the master database:

```
User.create(username='Peewee')  # Executed against master.
```

When you execute a read query, it will run against one of the replicas:

```
users = User.select().where(User.username == 'Peewee')
```

Note: To force a `SELECT` query against the master database, manually create the *`SelectQuery`*.

```
SelectQuery(User)  # master database.
```

Note: Queries will be dispatched among the `read_slaves` in round-robin fashion.

Test Utils

Contains utilities helpful when testing peewee projects.

class **`test_database`**(*db*, *models*[, *create_tables=True*[, *fail_silently=False*]])

Context manager that lets you use a different database with a set of models. Models can also be automatically created and dropped.

This context manager helps make it possible to test your peewee models using a "test-only" database.

Parameters

- **`db`** (`Database`) – Database to use with the given models
- **`models`** – a `list` or `tuple` of *`Model`* classes to use with the `db`
- **`create_tables`** (*`boolean`*) – Whether tables should be automatically created and dropped.
- **`fail_silently`** (*`boolean`*) – Whether the table create / drop should fail silently.

Example:

```
from unittest import TestCase
from playhouse.test_utils import test_database
from peewee import *

from my_app.models import User, Tweet

test_db = SqliteDatabase(':memory:')

class TestUsersTweets(TestCase):
    def create_test_data(self):
        # ... create a bunch of users and tweets
        for i in range(10):
            User.create(username='user-%d' % i)
```

```
    def test_timeline(self):
        with test_database(test_db, (User, Tweet)):
            # This data will be created in `test_db`
            self.create_test_data()

            # Perform assertions on test data inside ctx manager.
            self.assertEqual(Tweet.timeline('user-0') [...])

        with test_database(test_db, (User,)):
            # Test something that just affects user.
            self.test_some_user_thing()

        # once we exit the context manager, we're back to using the normal␣
↪database
```

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()
```

pskel

I often find myself writing very small scripts with peewee. *pskel* will generate the boilerplate code for a basic peewee script.

Usage:

```
pskel [options] model1 model2 ...
```

pskel accepts the following options:

Option	Default	Description
`-l,--logging`	False	Log all queries to stdout.
`-e,--engine`	sqlite	Database driver to use.
`-d,--database`	`:memory:`	Database to connect to.

Example:

```
$ pskel -e postgres -d my_database User Tweet
```

This will print the following code to *stdout* (which you can redirect into a file using >):

```
#!/usr/bin/env python

import logging

from peewee import *
from peewee import create_model_tables

db = PostgresqlDatabase('my_database')

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

class User(BaseModel):
    pass

class Tweet(BaseModel):
    pass

def main():
    create_model_tables([User, Tweet], fail_silently=True)

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

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 it's 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):
    username = CharField(unique=True)

class Tweet(db_wrapper.Model):
    user = ForeignKeyField(User, related_name='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*)

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.

Parameters

- **query_or_model** – Either a *Model* class or a pre-filtered *SelectQuery*.
- **query** – An arbitrarily complex peewee expression.

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*]]]]])

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.

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.

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*]])

Helper class to perform pagination based on `GET` arguments.

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.

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.

API Reference

Models

class Model(***kwargs*)
Models provide a 1-to-1 mapping to database tables. 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.

Parameters **kwargs** – Initialize the model, assigning the given key/values to the appropriate fields.

Example:

```
class User(Model):
    username = CharField()
    join_date = DateTimeField(default=datetime.datetime.now)
    is_admin = BooleanField()

u = User(username='charlie', is_admin=True)
```

classmethod select(**selection*)

Parameters **selection** – A list of model classes, field instances, functions or expressions. If no argument is provided, all columns for the given model will be selected.

Return type a *SelectQuery* for the given *Model*.

Examples of selecting all columns (default):

```
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):

```
(Tweet
  .select(Tweet, User)
  .join(User)
  .order_by(Tweet.created_date.desc()))
```

classmethod update(***update*)

Parameters **update** – mapping of field-name to expression

Return type an *UpdateQuery* for the given *Model*

Example showing users being marked inactive if their registration expired:

```
q = User.update(active=False).where(User.registration_expired == True)
q.execute()  # Execute the query, updating the database.
```

Example showing an atomic update:

```
q = PageView.update(count=PageView.count + 1).where(PageView.url == url)
q.execute()  # execute the query, updating the database.
```

Note: When an update query is executed, the number of rows modified will be returned.

classmethod insert(***insert*)

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.

Parameters **insert** – mapping of field or field-name to expression.

Return type an *InsertQuery* for the given *Model*.

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:

```
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.

insert_many(*rows*)

Insert multiple rows at once. The `rows` parameter must be an iterable that yields dictionaries. 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"!
])
```

Parameters **rows** – An iterable containing dictionaries of field-name-to-value.

Return type an *InsertQuery* for the given *Model*.

Example of inserting multiple Users:

```
usernames = ['charlie', 'huey', 'peewee', 'mickey']
row_dicts = ({'username': username} for username in usernames)

# Insert 4 new rows.
User.insert_many(row_dicts).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

classmethod **insert_from**(*fields*, *query*)

Insert rows into the table using a query as the data source. This API should be used for *INSERT INTO...SELECT FROM* queries.

Parameters

- **fields** – The field objects to map the selected data into.
- **query** – The source of the new rows.

Return type an *InsertQuery* for the given *Model*.

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(
    [UserTweetDenorm.username, UserTweetDenorm.num_tweets],
    source).execute()
```

classmethod delete()

Return type a *DeleteQuery* for the given *Model*.

Example showing the deletion of all inactive users:

```
q = User.delete().where(User.active == False)
q.execute()  # remove the rows
```

Warning: This method performs a delete on the *entire table*. To delete a single instance, see *Model.delete_instance()*.

classmethod raw(*sql*, **params*)

Parameters

- **sql** – a string SQL expression
- **params** – any number of parameters to interpolate

Return type a *RawQuery* for the given Model

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 create(***attributes*)

Parameters **attributes** – key/value pairs of model attributes

Return type a model instance with the provided attributes

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 get(**args*)

Parameters **args** – a list of query expressions, e.g. User.username == 'foo'

Return type *Model* instance or raises DoesNotExist exception

Get a single row from the database that matches the given query. Raises a <model-class>.DoesNotExist if no rows are returned:

```
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_create([*defaults=None*[, ***kwargs*]])

Parameters

- **defaults** (*dict*) – A dictionary of values to set on newly-created model instances.
- **kwargs** – Django-style filters specifying which model to get, and what values to apply to new instances.

Returns A 2-tuple containing the model instance and a boolean indicating whether the instance was created.

This function attempts to retrieve a model instance based on the provided filters. If no matching model can be found, a new model is created using the parameters specified by the filters and any values in the `defaults` dictionary.

Note: Use care when calling `get_or_create` with `autocommit=False`, as the `get_or_create()` method will call *Database.atomic()* to create either a transaction or savepoint.

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 alias()

Return type `ModelAlias` instance

The *alias()* method is used to create self-joins.

Example:

```
Parent = Category.alias()
sq = (Category
      .select(Category, Parent)
      .join(Parent, on=(Category.parent == Parent.id))
      .where(Parent.name == 'parent category'))
```

Note: When using a `ModelAlias` in a join, you must explicitly specify the join condition.

classmethod **create_table**([*fail_silently=False*])

Parameters **fail_silently** (*bool*) – If set to `True`, the method will check for the existence of the table before attempting to create.

Create the table for the given model, along with any constraints and indexes.

Example:

```
database.connect()
SomeModel.create_table()  # Execute the create table query.
```

classmethod **drop_table**([*fail_silently=False*[, *cascade=False*]])

Parameters

- **fail_silently** (*bool*) – If set to `True`, the query will check for the existence of the table before attempting to remove.
- **cascade** (*bool*) – Drop table with `CASCADE` option.

Drop the table for the given model.

classmethod **table_exists**()

Return type Boolean whether the table for this model exists in the database

classmethod **sqlall**()

Returns A list of queries required to create the table and indexes.

save([*force_insert=False*[, *only=None*]])

Parameters

- **force_insert** (*bool*) – Whether to force execution of an insert
- **only** (*list*) – A list of fields to persist – when supplied, only the given fields will be persisted.

Save the given instance, creating or updating depending on whether it has a primary key. If `force_insert=True` an *INSERT* will be issued regardless of whether or not the primary key exists.

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
```

delete_instance([*recursive=False*[, *delete_nullable=False*]])

Parameters

- **recursive** – Delete this instance and anything that depends on it, optionally updating those that have nullable dependencies
- **delete_nullable** – If doing a recursive delete, delete all dependent objects regardless of whether it could be updated to NULL

Delete the given instance. Any foreign keys set to cascade on delete will be deleted automatically. For more programmatic control, you can call with 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
```

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.

dirty_fields

Return a list of fields that were manually set.

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
```

---

is_dirty()

Return whether any fields were manually set.

Return type bool

prepared()

This method provides a hook for performing model initialization *after* the row data has been populated.

Fields

Field(null=False, index=False, unique=False, verbose_name=None, help_text=None, db_column=N

The base class from which all other field types extend.

Parameters

- **null** (*bool*) – whether this column can accept `None` or `NULL` values
- **index** (*bool*) – whether to create an index for this column when creating the table
- **unique** (*bool*) – whether to create a unique index for this column when creating the table
- **verbose_name** (*string*) – specify a "verbose name" for this field, useful for metadata purposes
- **help_text** (*string*) – specify some instruction text for the usage/meaning of this field
- **db_column** (*string*) – column name to use for underlying storage, useful for compatibility with legacy databases
- **default** – a value to use as an uninitialized default
- **choices** – an iterable of 2-tuples mapping `value` to `display`
- **primary_key** (*bool*) – whether to use this as the primary key for the table
- **sequence** (*string*) – name of sequence (if backend supports it)
- **constraints** (*list*) – a list of constraints, e.g. `[Check('price > 0')]`.
- **schema** (*string*) – name of schema (if backend supports it)
- **kwargs** – named attributes containing values that may pertain to specific field subclasses, such as "max_length" or "decimal_places"

db_field = '<some field type>'
Attribute used to map this field to a column type, e.g. "string" or "datetime"

_is_bound
Boolean flag indicating if the field is attached to a model class.

model_class
The model the field belongs to. *Only applies to bound fields.*

name
The name of the field. *Only applies to bound fields.*

db_value(*value*)

Parameters **value** – python data type to prep for storage in the database

Return type converted python datatype

python_value(*value*)

Parameters **value** – data coming from the backend storage

Return type python data type

coerce(*value*)
This method is a shorthand that is used, by default, by both `db_value` and `python_value`. You can usually get away with just implementing this.

Parameters **value** – arbitrary data from app or backend

Return type python data type

class IntegerField
Stores: integers

db_field = 'int'

class BigIntegerField
Stores: big integers

db_field = 'bigint'

class PrimaryKeyField
Stores: auto-incrementing integer fields suitable for use as primary key.

db_field = 'primary_key'

class FloatField
Stores: floating-point numbers

db_field = 'float'

class DoubleField
Stores: double-precision floating-point numbers

db_field = 'double'

class DecimalField
Stores: decimal numbers, using python standard library `Decimal` objects

Additional attributes and values:

max_digits	10
decimal_places	5
auto_round	False
rounding	decimal.DefaultContext.rounding

db_field = 'decimal'

class CharField
Stores: small strings (0-255 bytes)

Additional attributes and values:

max_length	255

db_field = 'string'

class TextField
Stores: arbitrarily large strings

db_field = 'text'

class DateTimeField
Stores: python `datetime.datetime` instances

Accepts a special parameter `formats`, which contains a list of formats the datetime can be encoded with. The default behavior is:

```
'%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: If the incoming value does not match a format, it will be returned as-is

---

db_field = 'datetime'

year
An expression suitable for extracting the year, for example to retrieve all blog posts from 2013:

```
Blog.select().where(Blog.pub_date.year == 2013)
```

month
An expression suitable for extracting the month from a stored date.

day
An expression suitable for extracting the day from a stored date.

hour
An expression suitable for extracting the hour from a stored time.

minute
An expression suitable for extracting the minute from a stored time.

second
An expression suitable for extracting the second from a stored time.

class DateField
Stores: python `datetime.date` instances

Accepts a special parameter `formats`, which contains a list of formats the date can be encoded with. The default behavior is:

```
'%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 will be returned as-is

db_field = 'date'

year
An expression suitable for extracting the year, for example to retrieve all people born in 1980:

```
Person.select().where(Person.dob.year == 1983)
```

month
Same as *year*, except extract month.

day
Same as *year*, except extract day.

class TimeField
Stores: python `datetime.time` instances

Accepts a special parameter `formats`, which contains a list of formats the time can be encoded with. The default behavior is:

```
'%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 will be returned as-is

db_field = 'time'

hour
Extract the hour from a time, for example to retreive all events occurring in the evening:

```
Event.select().where(Event.time.hour > 17)
```

minute
Same as *hour*, except extract minute.

second
Same as *hour*, except extract second..

class TimestampField
Stores: python `datetime.datetime` instances (stored as integers)

Accepts a special parameter `resolution`, which is a power-of-10 up to `10^6`. This allows sub-second precision while still using an *IntegerField* for storage. Default is `1` (second precision).

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 BooleanField
Stores: `True` / `False`

db_field = 'bool'

class BlobField
Store arbitrary binary data.

class UUIDField
Store `UUID` values.

Note: Currently this field is only supported by *PostgresqlDatabase*.

---

class BareField
Intended to be used only with SQLite. 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.

Note: Currently this field is only supported by *SqliteDatabase*.

---

class ForeignKeyField(*rel_model*[, *related_name=None*[, *on_delete=None*[, *on_update=None*[, *to_field=None*[, ...]]]]])
Stores: relationship to another model

Parameters

- **rel_model** – related *Model* class or the string 'self' if declaring a self-referential foreign key
- **related_name** (*string*) – attribute to expose on related model
- **on_delete** (*string*) – on delete behavior, e.g. `on_delete='CASCADE'`.
- **on_update** (*string*) – on update behavior.

- **to_field** – the field (or field name) on `rel_model` the foreign key references. Defaults to the primary key field for `rel_model`.

```
class User(Model):
    name = CharField()

class Tweet(Model):
    user = ForeignKeyField(User, related_name='tweets')
    content = TextField()

# "user" attribute
>>> some_tweet.user
<User: charlie>

# "tweets" related name attribute
>>> for tweet in charlie.tweets:
...     print tweet.content
Some tweet
Another tweet
Yet another tweet
```

Note: Foreign keys do not have a particular `db_field` 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 `to_field`, that field must be either a primary key or have a unique constraint.

class CompositeKey(**fields*)

Specify a composite primary key for a model. 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, related_name='tags')
    tag = ForeignKeyField(Tag, related_name='blogs')

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

Query Types

class Query

The parent class from which all other query classes are derived. While you will not deal with *Query* directly in your code, it implements some methods that are common across all query types.

where(**expressions*)

Parameters **expressions** – a list of one or more expressions

Return type a *Query* instance

Example selection users where the username is equal to 'somebody':

```
sq = SelectQuery(User).where(User.username == 'somebody')
```

Example selecting tweets made by users who are either editors or administrators:

```
sq = SelectQuery(Tweet).join(User).where(
    (User.is_editor == True) |
    (User.is_admin == True))
```

Example of deleting tweets by users who are no longer active:

```
dq = DeleteQuery(Tweet).where(
    Tweet.user << User.select().where(User.active == False))
dq.execute()  # perform the delete query
```

Note: `where()` calls are chainable. Multiple calls will be "AND"-ed together.

join(*model*, *join_type=None*, *on=None*)

Parameters

- **model** – the model to join on. there must be a *ForeignKeyField* between the current `query context` and the model passed in.
- **join_type** – allows the type of `JOIN` used to be specified explicitly, one of `JOIN.INNER`, `JOIN.LEFT_OUTER`, `JOIN.FULL`, `JOIN.RIGHT_OUTER`, or `JOIN.CROSS`.
- **on** – if multiple foreign keys exist between two models, this parameter is the ForeignKeyField to join on.

Return type a *Query* instance

Generate a `JOIN` clause from the current `query context` to the `model` passed in, and establishes `model` as the new `query context`.

Example selecting tweets and joining on user in order to restrict to only those tweets made by "admin" users:

```
sq = SelectQuery(Tweet).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 = SelectQuery(User).join(Relationship, on=Relationship.to_user)
```

switch(*model*)

Parameters **model** – model to switch the `query context` to.

Return type a clone of the query with a new query context

Switches the `query context` to the given model. Raises an exception if the model has not been selected or joined on previously. Useful for performing multiple joins from a single table.

The following example selects from blog and joins on both entry and user:

```
sq = SelectQuery(Blog).join(Entry).switch(Blog).join(User)
```

alias(*alias=None*)

Parameters **alias** (*str*) – A string to alias the result of this query

Return type a Query instance

Assign an alias to given query, which can be used as part of a subquery.

sql()

Return type a 2-tuple containing the appropriate SQL query and a tuple of parameters

execute()

Execute the given query

scalar([*as_tuple=False*[, *convert=False*]])

Parameters

- **as_tuple** (*bool*) – return the row as a tuple or a single value
- **convert** (*bool*) – attempt to coerce the selected value to the appropriate data-type based on it's associated Field type (assuming one exists).

Return type the resulting row, either as a single value or tuple

Provide a way to retrieve single values from select queries, for instance when performing an aggregation.

```
>>> PageView.select(fn.Count(fn.Distinct(PageView.url))).scalar()
100 # <-- there are 100 distinct URLs in the pageview table
```

This example illustrates the use of the *convert* argument. When using a SQLite database, datetimes are stored as strings. To select the max datetime, and have it *returned* as a datetime, we will specify `convert=True`.

```
>>> PageView.select(fn.MAX(PageView.timestamp)).scalar()
'2016-04-20 13:37:00.1234'

>>> PageView.select(fn.MAX(PageView.timestamp)).scalar(convert=True)
datetime.datetime(2016, 4, 20, 13, 37, 0, 1234)
```

class SelectQuery(*model_class*, **selection*)

By far the most complex of the query classes available in peewee. It supports all clauses commonly associated with select queries.

Methods on the select query can be chained together.

`SelectQuery` implements an *__iter__()* method, allowing it to be iterated to return model instances.

Parameters

- **model** – a *Model* class to perform query on
- **selection** – a list of models, fields, functions or expressions

If no selection is provided, it will default to all the fields of the given model.

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:

```
sq = SelectQuery(User, User.id, User.username)
for user in sq:
    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:

```
sq = (SelectQuery(
    User, User, fn.Count(Tweet.id).alias('count'))
```

```
    .join(Tweet)
    .group_by(User))
```

select(**selection*)

Parameters **selection** – a list of expressions, which can be model classes or fields. if left blank, will default to all the fields of the given model.

Return type *SelectQuery*

Note: Usually the selection will be specified when the instance is created. This method simply exists for the case when you want to modify the SELECT clause independent of instantiating a query.

```
query = User.select()
query = query.select(User.username)
```

from_(**args*)

Parameters **args** – one or more expressions, for example *Model* or *SelectQuery* instance(s). if left blank, will default to the table of the given model.

Return type *SelectQuery*

```
# rather than a join, select from both tables and join with where.
query = User.select().from_(User, Blog).where(Blog.user == User.id)
```

group_by(**clauses*)

Parameters **clauses** – a list of expressions, which can be model classes or individual field instances

Return type *SelectQuery*

Group by one or more columns. If a model class is provided, all the fields on that model class will be used.

Example selecting users, joining on tweets, and grouping by the user so a count of tweets can be calculated for each user:

```
sq = (User
    .select(User, fn.Count(Tweet.id).alias('count'))
    .join(Tweet)
    .group_by(User))
```

having(**expressions*)

Parameters **expressions** – a list of one or more expressions

Return type *SelectQuery*

Here is the above example selecting users and tweet counts, but restricting the results to those users who have created 100 or more tweets:

```
sq = (User
    .select(User, fn.Count(Tweet.id).alias('count'))
    .join(Tweet)
    .group_by(User)
    .having(fn.Count(Tweet.id) > 100))
```

order_by(**clauses*[, *extend=False*])

Parameters

- **clauses** – a list of fields, calls to `field.[asc|desc]()` or one or more expressions. If called without any arguments, any pre-existing `ORDER BY` clause will be removed.
- **extend** – When called with `extend=True`, Peewee will append any to the pre-existing `ORDER BY` rather than overwriting it.

Return type *SelectQuery*

Example of ordering users by username:

```
User.select().order_by(User.username)
```

Example of selecting tweets and ordering them first by user, then newest first:

```
query = (Tweet
         .select()
         .join(User)
         .order_by(
             User.username,
             Tweet.created_date.desc()))
```

You can also use + and − prefixes to indicate ascending or descending order if you prefer:

```
query = (Tweet
         .select()
         .join(User)
         .order_by(
             +User.username,
             -Tweet.created_date))
```

A more complex example ordering users by the number of tweets made (greatest to least), then ordered by username in the event of a tie:

```
tweet_ct = fn.Count(Tweet.id)
sq = (User
    .select(User, tweet_ct.alias('count'))
    .join(Tweet)
    .group_by(User)
    .order_by(tweet_ct.desc(), User.username))
```

Example of removing a pre-existing `ORDER BY` clause:

```
# Query will be ordered by username.
users = User.select().order_by(User.username)

# Query will be returned in whatever order database chooses.
unordered_users = users.order_by()
```

window(**windows*)

Parameters **windows** (Window) – One or more *Window* instances.

Add one or more window definitions to this query.

```
window = Window(partition_by=[fn.date_trunc('day', PageView.timestamp)])
query = (PageView
         .select(
             PageView.url,
```

```
            PageView.timestamp,
            fn.Count(PageView.id).over(window=window))
        .window(window)
        .order_by(PageView.timestamp))
```

limit(*num*)

Parameters **num** (*int*) – limit results to num rows

offset(*num*)

Parameters **num** (*int*) – offset results by num rows

paginate(*page_num*, *paginate_by=20*)

Parameters

- **page_num** – a 1-based page number to use for paginating results
- **paginate_by** – number of results to return per-page

Return type *SelectQuery*

Shorthand for applying a LIMIT and OFFSET to the query.

Page indices are **1-based**, so page 1 is the first page.

```
User.select().order_by(User.username).paginate(3, 20)  # get users 41-60
```

distinct([*is_distinct=True*])

Parameters **is_distinct** – See notes.

Return type *SelectQuery*

Indicates that this query should only return distinct rows. Results in a SELECT DISTINCT query.

Note: The value for is_distinct should either be a boolean, in which case the query will (or won't) be *DISTINCT*.

You can specify a list of one or more expressions to generate a DISTINCT ON query, e.g. .distinct([Model.col1, Model.col2]).

---

for_update([*for_update=True*[, *nowait=False*]])

Return type *SelectQuery*

Indicate that this query should lock rows for update. If nowait is True then the database will raise an OperationalError if it cannot obtain the lock.

with_lock([*lock_type='UPDATE'*])

Return type *SelectQuery*

Indicates that this query shoudl lock rows. A more generic version of the *for_update()* method.

Example:

```
# SELECT * FROM some_model FOR KEY SHARE NOWAIT;
SomeModel.select().with_lock('KEY SHARE NOWAIT')
```

Note: You do not need to include the word *FOR*.

naive()

Return type *SelectQuery*

Flag this query indicating it should only attempt to reconstruct a single model instance for every row returned by the cursor. If multiple tables were queried, the columns returned are patched directly onto the single model instance.

Generally this method is useful for speeding up the time needed to construct model instances given a database cursor.

Note: this can provide a significant speed improvement when doing simple iteration over a large result set.

iterator()

Return type iterable

By default peewee will cache rows returned by the cursor. This is to prevent things like multiple iterations, slicing and indexing from triggering extra queries. When you are iterating over a large number of rows, however, this cache can take up a lot of memory. Using `iterator()` will save memory by not storing all the returned model instances.

```
# iterate over large number of rows.
for obj in Stats.select().iterator():
    # do something.
    pass
```

tuples()

Return type *SelectQuery*

Flag this query indicating it should simply return raw tuples from the cursor. This method is useful when you either do not want or do not need full model instances.

dicts()

Return type *SelectQuery*

Flag this query indicating it should simply return dictionaries from the cursor. This method is useful when you either do not want or do not need full model instances.

aggregate_rows()

Return type *SelectQuery*

This method provides one way to avoid the **N+1** query problem.

Consider a webpage where you wish to display a list of users and all of their associated tweets. You could approach this problem by listing the users, then for each user executing a separate query to retrieve their tweets. This is the **N+1** behavior, because the number of queries varies depending on the number of users. Conventional wisdom is that it is preferable to execute fewer queries. Peewee provides several ways to avoid this problem.

You can use the *prefetch()* helper, which uses `IN` clauses to retrieve the tweets for the listed users.

Another method is to select both the user and the tweet data in a single query, then de-dupe the users, aggregating the tweets in the process.

The raw column data might appear like this:

```
# user.id, user.username, tweet.id, tweet.user_id, tweet.message
[1,         'charlie',     1,         1,              'hello'],
[1,         'charlie',     2,         1,              'goodbye'],
[2,         'no-tweets',   NULL,      NULL,           NULL],
[3,         'huey',        3,         3,              'meow'],
[3,         'huey',        4,         3,              'purr'],
[3,         'huey',        5,         3,              'hiss'],
```

We can infer from the `JOIN` clause that the user data will be duplicated, and therefore by de-duping the users, we can collect their tweets in one go and iterate over the users and tweets transparently.

```
query = (User
         .select(User, Tweet)
         .join(Tweet, JOIN.LEFT_OUTER)
         .order_by(User.username, Tweet.id)
         .aggregate_rows())  # .aggregate_rows() tells peewee to de-dupe the␣
↪rows.
for user in query:
    print user.username
    for tweet in user.tweets:
        print '  ', tweet.message

# Producing the following output:
charlie
   hello
   goodbye
huey
   meow
   purr
   hiss
no-tweets
```

Warning: Be sure that you specify an `ORDER BY` clause that ensures duplicated data will appear in consecutive rows.

Note: You can specify arbitrarily complex joins, though for more complex queries it may be more efficient to use *`prefetch()`*. In short, try both and see what works best for your data-set.

Note: For more information, see the *Avoiding N+1 queries* document and the *Using aggregate_rows* sub-section.

annotate(*related_model*, *aggregation=None*)

Parameters

- **related_model** – related *`Model`* on which to perform aggregation, must be linked by *`ForeignKeyField`*.
- **aggregation** – the type of aggregation to use, e.g. `fn.Count(Tweet.id).alias('count')`

Return type *`SelectQuery`*

Annotate a query with an aggregation performed on a related model, for example, "get a list of users with the number of tweets for each":

```
>>> User.select().annotate(Tweet)
```

If `aggregation` is None, it will default to `fn.Count(related_model.id).alias('count')` but can be anything:

```
>>> user_latest = User.select().annotate(Tweet, fn.Max(Tweet.created_date).
↪alias('latest'))
```

Note: If the `ForeignKeyField` is `nullable`, then a `LEFT OUTER` join may need to be used:

```
query = (User
         .select()
         .join(Tweet, JOIN.LEFT_OUTER)
         .switch(User)  # Switch query context back to `User`.
         .annotate(Tweet))
```

---

aggregate(*aggregation*)

Parameters **aggregation** – a function specifying what aggregation to perform, for example `fn.Max(Tweet.created_date)`.

Method to look at an aggregate of rows using a given function and return a scalar value, such as the count of all rows or the average value of a particular column.

count([*clear_limit=False*])

Parameters **clear_limit** (*bool*) – Remove any limit or offset clauses from the query before counting.

Return type an integer representing the number of rows in the current query

---

Note: If the query has a GROUP BY, DISTINCT, LIMIT, or OFFSET clause, then the *wrapped_count()* method will be used instead.

```
>>> sq = SelectQuery(Tweet)
>>> sq.count()
45  # number of tweets
>>> deleted_tweets = sq.where(Tweet.status == DELETED)
>>> deleted_tweets.count()
3  # number of tweets that are marked as deleted
```

wrapped_count([*clear_limit=False*])

Parameters **clear_limit** (*bool*) – Remove any limit or offset clauses from the query before counting.

Return type an integer representing the number of rows in the current query

Wrap the count query in a subquery. Additional overhead but will give correct counts when performing `DISTINCT` queries or those with `GROUP BY` clauses.

Note: *count()* will automatically default to *wrapped_count()* in the event the query is distinct or

has a grouping.

---

exists()

Return type boolean whether the current query will return any rows. uses an optimized lookup, so use this rather than *get()*.

```
sq = User.select().where(User.active == True)
if sq.where(User.username == username, User.active == True).exists():
    authenticated = True
```

get()

Return type *Model* instance or raises DoesNotExist exception

Get a single row from the database that matches the given query. Raises a <model-class>.DoesNotExist if no rows are returned:

```
active = User.select().where(User.active == True)
try:
    user = active.where(User.username == username).get()
except User.DoesNotExist:
    user = None
```

This method is also exposed via the *Model* api, in which case it accepts arguments that are translated to the where clause:

user = User.get(User.active == True, User.username == username)

first([*n=1*])

Parameters **n** (*int*) – Return the first *n* query results after applying a limit of n records.

Return type *Model* instance, list or None if no results

Fetch the first *n* rows from a query. Behind-the-scenes, a LIMIT n is applied. The results of the query are then cached on the query result wrapper so subsequent calls to *first()* will not cause multiple queries.

If only one row is requested (default behavior), then the return-type will be either a model instance or None.

If multiple rows are requested, the return type will either be a list of one to n model instances, or None if no results are found.

peek([*n=1*])

Parameters **n** (*int*) – Return the first *n* query results.

Return type *Model* instance, list or None if no results

Fetch the first *n* rows from a query. No LIMIT is applied to the query, so the *peek()* has slightly different semantics from *first()*, which ensures no more than *n* rows are requested. The peek method, on the other hand, retains the ability to fetch the entire result set withouth issuing additional queries.

execute()

Return type QueryResultWrapper

Executes the query and returns a QueryResultWrapper for iterating over the result set. The results are managed internally by the query and whenever a clause is added that would possibly alter the result set, the query is marked for re-execution.

__iter__()

Executes the query and returns populated model instances:

```
for user in User.select().where(User.active == True):
    print user.username
```

__len__()

Return the number of items in the result set of this query. If all you need is the count of items and do not intend to do anything with the results, call *count()*.

> **Warning:** The SELECT query will be executed and the result set will be loaded. If you want to obtain the number of results without also loading the query, use *count()*.

__getitem__(*value*)

Parameters **value** – Either an index or a slice object.

Return the model instance(s) at the requested indices. To get the first model, for instance:

```
query = User.select().order_by(User.username)
first_user = query[0]
first_five = query[:5]
```

__or__(*rhs*)

Parameters **rhs** – Either a *SelectQuery* or a *CompoundSelect*

Return type *CompoundSelect*

Create a UNION query with the right-hand object. The result will contain all values from both the left and right queries.

```
customers = Customer.select(Customer.city).where(Customer.state == 'KS')
stores = Store.select(Store.city).where(Store.state == 'KS')

# Get all cities in kansas where we have either a customer or a store.
all_cities = (customers | stores).order_by(SQL('city'))
```

> **Note:** SQLite does not allow ORDER BY or LIMIT clauses on the components of a compound query, however SQLite does allow these clauses on the final, compound result. This applies to UNION (ALL), INTERSECT, and EXCEPT.

__and__(*rhs*)

Parameters **rhs** – Either a *SelectQuery* or a *CompoundSelect*

Return type *CompoundSelect*

Create an INTERSECT query. The result will contain values that are in both the left and right queries.

```
customers = Customer.select(Customer.city).where(Customer.state == 'KS')
stores = Store.select(Store.city).where(Store.state == 'KS')

# Get all cities in kanasas where we have both customers and stores.
cities = (customers & stores).order_by(SQL('city'))
```

__sub__(*rhs*)

Parameters **rhs** – Either a *SelectQuery* or a *CompoundSelect*

Return type *CompoundSelect*

Create an EXCEPT query. The result will contain values that are in the left-hand query but not in the right-hand query.

```
customers = Customer.select(Customer.city).where(Customer.state == 'KS')
stores = Store.select(Store.city).where(Store.state == 'KS')

# Get all cities in kanasas where we have customers but no stores.
cities = (customers - stores).order_by(SQL('city'))
```

__xor__(*rhs*)

Parameters **rhs** – Either a *SelectQuery* or a *CompoundSelect*

Return type *CompoundSelect*

Create an symmetric difference query. The result will contain values that are in either the left-hand query or the right-hand query, but not both.

```
customers = Customer.select(Customer.city).where(Customer.state == 'KS')
stores = Store.select(Store.city).where(Store.state == 'KS')

# Get all cities in kanasas where we have either customers with no
# store, or a store with no customers.
cities = (customers ^ stores).order_by(SQL('city'))
```

class UpdateQuery(*model_class*, ***kwargs*)

Parameters

- **model** – *Model* class on which to perform update
- **kwargs** – mapping of field/value pairs containing columns and values to update

Example in which users are marked inactive if their registration expired:

```
uq = UpdateQuery(User, active=False).where(User.registration_expired == True)
uq.execute()  # Perform the actual update
```

Example of an atomic update:

```
atomic_update = UpdateQuery(PageCount, count = PageCount.count + 1).where(
    PageCount.url == url)
atomic_update.execute()  # will perform the actual update
```

execute()

Return type Number of rows updated

Performs the query

returning(**returning*)

Parameters **returning** – A list of model classes, field instances, functions or expressions. If no argument is provided, all columns for the given model will be selected. To clear any existing values, pass in None.

Return type a *UpdateQuery* for the given *Model*.

Add a `RETURNING` clause to the query, which will cause the `UPDATE` to compute return values based on each row that was actually updated.

When the query is executed, rather than returning the number of rows updated, an iterator will be returned that yields the updated objects.

Note: Currently only *PostgresqlDatabase* supports this feature.

Example:

```
# Disable all users whose registration expired, and return the user
# objects that were updated.
query = (User
         .update(active=False)
         .where(User.registration_expired == True)
         .returning(User))

# We can iterate over the users that were updated.
for updated_user in query.execute():
    send_activation_email(updated_user.email)
```

For more information, check out *the RETURNING clause docs*.

tuples()

Return type *UpdateQuery*

Note: This method should only be used in conjunction with a call to *returning()*.

When the updated results are returned, they will be returned as row tuples.

dicts()

Return type *UpdateQuery*

Note: This method should only be used in conjunction with a call to *returning()*.

When the updated results are returned, they will be returned as dictionaries mapping column to value.

on_conflict([*action=None*])

Add a SQL `ON CONFLICT` clause with the specified action to the given `UPDATE` query. Valid actions are:

- ROLLBACK
- ABORT
- FAIL
- IGNORE
- REPLACE

Specifying `None` for the action will execute a normal `UPDATE` query.

Note: This feature is only available on SQLite databases.

class **InsertQuery**(*model_class*[, *field_dict=None*[, *rows=None*[, *fields=None*[, *query=None*[, *validate_fields=False*]]]]])

Creates an `InsertQuery` instance for the given model.

Parameters

- **field_dict** (*dict*) – A mapping of either field or field-name to value.
- **rows** (*iterable*) – An iterable of dictionaries containing a mapping of field or field-name to value.
- **fields** (*list*) – A list of field objects to insert data into (only used in combination with the `query` parameter).
- **query** – A *SelectQuery* to use as the source of data.
- **validate_fields** (*bool*) – Check that every column referenced in the insert query has a corresponding field on the model. If validation is enabled and then fails, a `KeyError` is raised.

Basic example:

```
>>> fields = {'username': 'admin', 'password': 'test', 'active': True}
>>> iq = InsertQuery(User, fields)
>>> iq.execute()  # insert new row and return primary key
2L
```

Example inserting multiple rows:

```
users = [
    {'username': 'charlie', 'active': True},
    {'username': 'peewee', 'active': False},
    {'username': 'huey', 'active': True}]
iq = InsertQuery(User, rows=users)
iq.execute()
```

Example inserting using a query as the data source:

```
query = (User
         .select(User.username, fn.COUNT(Tweet.id))
         .join(Tweet, JOIN.LEFT_OUTER)
         .group_by(User.username))
iq = InsertQuery(
    UserTweetDenorm,
    fields=[UserTweetDenorm.username, UserTweetDenorm.num_tweets],
    query=query)
iq.execute()
```

execute()

Return type primary key of the new row

Performs the query

upsert([*upsert=True*])

Perform an *INSERT OR REPLACE* query with SQLite. MySQL databases will issue a *REPLACE* query. Currently this feature is not supported for Postgres databases, but the 9.5 syntax will be added soon.

Note: This feature is only available on SQLite and MySQL databases.

---

on_conflict([*action=None*])

Add a SQL `ON CONFLICT` clause with the specified action to the given `INSERT` query. Specifying `REPLACE` is equivalent to using the *upsert()* method. Valid actions are:

- ROLLBACK
- ABORT
- FAIL
- IGNORE
- REPLACE

Specifying `None` for the action will execute a normal `INSERT` query.

Note: This feature is only available on SQLite databases.

---

return_id_list([*return_id_list=True*])

By default, when doing bulk INSERTs, peewee will not return the list of generated primary keys. However, if the database supports returning primary keys via `INSERT ... RETURNING`, this method instructs peewee to return the generated list of IDs.

Note: Currently only PostgreSQL supports this behavior. While other databases support bulk inserts, they will simply return `True` instead.

Example:

```
usernames = [
    {'username': username}
    for username in ['charlie', 'huey', 'mickey']]
query = User.insert_many(usernames).return_id_list()
user_ids = query.execute()
print user_ids
# prints something like [1, 2, 3]
```

returning(**returning*)

Parameters **returning** – A list of model classes, field instances, functions or expressions. If no argument is provided, all columns for the given model will be selected. To clear any existing values, pass in `None`.

Return type a *InsertQuery* for the given *Model*.

Add a `RETURNING` clause to the query, which will cause the `INSERT` to compute return values based on each row that was inserted.

When the query is executed, rather than returning the primary key of the new row(s), an iterator will be returned that yields the inserted objects.

Note: Currently only *PostgresqlDatabase* supports this feature.

Example:

```
# Create some users, retrieving the list of IDs assigned to them.
query = (User
         .insert_many(list_of_user_data)
```

```
        .returning(User))

# We can iterate over the users that were created.
for new_user in query.execute():
    # Do something with the new user's ID...
    do_something(new_user.id)
```

For more information, check out *the RETURNING clause docs*.

tuples()

Return type *InsertQuery*

Note: This method should only be used in conjunction with a call to *returning()*.

When the inserted results are returned, they will be returned as row tuples.

dicts()

Return type *InsertQuery*

Note: This method should only be used in conjunction with a call to *returning()*.

When the inserted results are returned, they will be returned as dictionaries mapping column to value.

class DeleteQuery(*model_class*)

Creates a *DELETE* query for the given model.

Note: DeleteQuery will *not* traverse foreign keys or ensure that constraints are obeyed, so use it with care.

Example deleting users whose account is inactive:

```
dq = DeleteQuery(User).where(User.active == False)
```

execute()

Return type Number of rows deleted

Performs the query

returning(**returning*)

Parameters **returning** – A list of model classes, field instances, functions or expressions. If no argument is provided, all columns for the given model will be selected. To clear any existing values, pass in `None`.

Return type a *DeleteQuery* for the given *Model*.

Add a `RETURNING` clause to the query, which will cause the `DELETE` to compute return values based on each row that was removed from the database.

When the query is executed, rather than returning the number of rows deleted, an iterator will be returned that yields the deleted objects.

Note: Currently only *PostgresqlDatabase* supports this feature.

Example:

```
# Create some users, retrieving the list of IDs assigned to them.
query = (User
         .delete()
         .where(User.account_expired == True)
         .returning(User))

# We can iterate over the user objects that were deleted.
for deleted_user in query.execute():
    # Do something with the deleted user.
    notify_account_deleted(deleted_user.email)
```

For more information, check out *the RETURNING clause docs*.

tuples()

Return type *DeleteQuery*

Note: This method should only be used in conjunction with a call to *returning()*.

When the deleted results are returned, they will be returned as row tuples.

dicts()

Return type *DeleteQuery*

Note: This method should only be used in conjunction with a call to *returning()*.

When the deleted results are returned, they will be returned as dictionaries mapping column to value.

class RawQuery(*model_class*, *sql*, **params*)

Allows execution of an arbitrary query and returns instances of the model via a `QueryResultsWrapper`.

Note: Generally you will only need this for executing highly optimized SELECT queries.

Warning: If you are executing a parameterized query, you must use the correct interpolation string for your database. SQLite uses `'?'` and most others use `'%s'`.

Example selecting users with a given username:

```
>>> rq = RawQuery(User, 'SELECT * FROM users WHERE username = ?', 'admin')
>>> for obj in rq.execute():
...     print obj
<User: admin>
```

tuples()

Return type *RawQuery*

Flag this query indicating it should simply return raw tuples from the cursor. This method is useful when you either do not want or do not need full model instances.

dicts()

Return type *RawQuery*

Flag this query indicating it should simply return raw dicts from the cursor. This method is useful when you either do not want or do not need full model instances.

execute()

Return type a `QueryResultWrapper` for iterating over the result set. The results are instances of the given model.

Performs the query

class CompoundSelect(*model_class*, *lhs*, *operator*, *rhs*)

Compound select query.

Parameters

- **model_class** – The type of model to return, by default the model class of the `lhs` query.
- **lhs** – Left-hand query, either a *SelectQuery* or a `CompoundQuery`.
- **operator** – A string used to join the two queries, for example `'UNION'`.
- **rhs** – Right query, either a *SelectQuery* or a `CompoundQuery`.

prefetch(*sq*, **subqueries*)

Parameters

- **sq** – *SelectQuery* instance
- **subqueries** – one or more *SelectQuery* instances to prefetch for `sq`. You can also pass models, but they will be converted into SelectQueries. If you wish to specify a particular model to join against, you can pass a 2-tuple of `(query_or_model, join_model)`.

Return type *SelectQuery* with related instances pre-populated

Pre-fetch the appropriate instances from the subqueries and apply them to their corresponding parent row in the outer query. This function will eagerly load the related instances specified in the subqueries. This is a technique used to save doing O(n) queries for n rows, and rather is O(k) queries for *k* subqueries.

For example, consider you have a list of users and want to display all their tweets:

```
# let's impost some small restrictions on our queries
users = User.select().where(User.active == True)
tweets = Tweet.select().where(Tweet.published == True)

# this will perform 2 queries
users_pf = prefetch(users, tweets)

# now we can:
for user in users_pf:
    print user.username
    for tweet in user.tweets_prefetch:
        print '- ', tweet.content
```

You can prefetch an arbitrary number of items. For instance, suppose we have a photo site, User -> Photo -> (Comments, Tags). That is, users can post photos, and these photos can have tags and comments on them. If we wanted to fetch a list of users, all their photos, and all the comments and tags on the photos:

```
users = User.select()
published_photos = Photo.select().where(Photo.published == True)
published_comments = Comment.select().where(
```

```
    (Comment.is_spam == False) &
    (Comment.num_flags < 3))

# note that we are just passing the Tag model -- it will be converted
# to a query automatically
users_pf = prefetch(users, published_photos, published_comments, Tag)

# now we can iterate users, photos, and comments/tags
for user in users_pf:
    for photo in user.photo_set_prefetch:
        for comment in photo.comment_set_prefetch:
            # ...
        for tag in photo.tag_set_prefetch:
            # ...
```

Note: Subqueries must be related by foreign key and can be arbitrarily deep

Note: For more information, see the *Avoiding N+1 queries* document and the *Using prefetch* sub-section.

Warning: *prefetch()* can use up lots of RAM when the result set is large, and will not warn you if you are doing something dangerous, so it is up to you to know when to use it. Additionally, because of the semantics of subquerying, there may be some cases when prefetch does not act as you expect (for instance, when applying a `LIMIT` to subqueries, but there may be others) – please report anything you think is a bug to github.

Database and its subclasses

class **Database**(*database*[, *threadlocals=True*[, *autocommit=True*[, *fields=None*[, *ops=None*[, *autorollback=False*[, *use_speedups=True*[, ***connect_kwargs*]]]]]]])

Parameters

- **database** – the name of the database (or filename if using sqlite)
- **threadlocals** (*bool*) – whether to store connections in a threadlocal
- **autocommit** (*bool*) – automatically commit every query executed by calling *execute()*
- **fields** (*dict*) – a mapping of `db_field` to database column type, e.g. 'string' => 'varchar'
- **ops** (*dict*) – a mapping of operations understood by the querycompiler to expressions
- **autorollback** (*bool*) – automatically rollback when an exception occurs while executing a query.
- **use_speedups** (*bool*) – use the Cython speedups module to improve performance of some queries.
- **connect_kwargs** – any arbitrary parameters to pass to the database driver when connecting

The `connect_kwargs` dictionary is used for vendor-specific parameters that will be passed back directly to your database driver, allowing you to specify the `user`, `host` and `password`, for instance. For more information and examples, see the *vendor-specific parameters document*.

Note: If your database name is not known when the class is declared, you can pass `None` in as the database name which will mark the database as "deferred" and any attempt to connect while in this state will raise an exception. To initialize your database, call the *`Database.init()`* method with the database name.

For an in-depth discussion of run-time database configuration, see the *Run-time database configuration* section.

A high-level API for working with the supported database engines. The database class:

- Manages the underlying database connection.
- Executes queries.
- Manage transactions and savepoints.
- Create and drop tables and indexes.
- Introspect the database.

`commit_select = False`
Whether to issue a commit after executing a select query. With some engines can prevent implicit transactions from piling up.

`compiler_class = QueryCompiler`
A class suitable for compiling queries

`compound_operations = ['UNION', 'INTERSECT', 'EXCEPT']`
Supported compound query operations.

`compound_select_parentheses = False`
Whether `UNION` (or other compound `SELECT` queries) allow parentheses around the queries.

`distinct_on = False`
Whether the database supports `DISTINCT ON` statements.

`drop_cascade = False`
Whether the database supports cascading drop table queries.

`field_overrides = {}`
A mapping of field types to database column types, e.g. `{'primary_key': 'SERIAL'}`

`foreign_keys = True`
Whether the given backend enforces foreign key constraints.

`for_update = False`
Whether the given backend supports selecting rows for update

`for_update_nowait = False`
Whether the given backend supports selecting rows for update

`insert_many = True`
Whether the database supports multiple `VALUES` clauses for `INSERT` queries.

`insert_returning = False`
Whether the database supports returning the primary key for newly inserted rows.

`interpolation = '?'`
The string used by the driver to interpolate query parameters

op_overrides = {}
A mapping of operation codes to string operations, e.g. `{OP.LIKE: 'LIKE BINARY'}`

quote_char = '"'
The string used by the driver to quote names

reserved_tables = []
Table names that are reserved by the backend – if encountered in the application a warning will be issued.

returning_clause = False
Whether the database supports `RETURNING` clauses for `UPDATE`, `INSERT` and `DELETE` queries.

Note: Currently only *PostgresqlDatabase* supports this.

See the following for more information:

- *UpdateQuery.returning()*
- *InsertQuery.returning()*
- *DeleteQuery.returning()*

savepoints = True
Whether the given backend supports savepoints.

sequences = False
Whether the given backend supports sequences

subquery_delete_same_table = True
Whether the given backend supports deleting rows using a subquery that selects from the same table

window_functions = False
Whether the given backend supports window functions.

init(*database*[, ***connect_kwargs*])
This method is used to initialize a deferred database. For details on configuring your database at run-time, see the *Run-time database configuration* section.

Parameters

- **database** – the name of the database (or filename if using sqlite)
- **connect_kwargs** – any arbitrary parameters to pass to the database driver when connecting

connect()
Establishes a connection to the database

Note: By default, connections will be stored on a threadlocal, ensuring connections are not shared across threads. To disable this behavior, initialize the database with `threadlocals=False`.

---

close()
Closes the connection to the database (if one is open)

Note: If you initialized with `threadlocals=True`, only a connection local to the calling thread will be closed.

---

initialize_connection(*conn*)
Perform additional intialization on a newly-opened connection. For example, if you are using SQLite you may want to enable foreign key constraint enforcement (off by default).

Here is how you might use this hook to load a SQLite extension:

```
class CustomSqliteDatabase(SqliteDatabase):
    def initialize_connection(self, conn):
        conn.load_extension('fts5')
```

get_conn()

Return type a connection to the database, creates one if does not exist

get_cursor()

Return type a cursor for executing queries

last_insert_id(*cursor*, *model*)

Parameters

- **cursor** – the database cursor used to perform the insert query
- **model** – the model class that was just created

Return type the primary key of the most recently inserted instance

rows_affected(*cursor*)

Return type number of rows affected by the last query

compiler()

Return type an instance of `QueryCompiler` using the field and op overrides specified.

execute(*clause*)

Parameters **clause** (`Node`) – a *Node* instance or subclass (e.g. a *SelectQuery*).

The clause will be compiled into SQL then sent to the *execute_sql()* method.

execute_sql(*sql*[, *params=None*[, *require_commit=True*]])

Parameters

- **sql** – a string sql query
- **params** – a list or tuple of parameters to interpolate

Note: You can configure whether queries will automatically commit by using the *set_autocommit()* and *Database.get_autocommit()* methods.

---

begin([*lock_type=None*])
Initiate a new transaction. By default **not** implemented as this is not part of the DB-API 2.0, but provided for API compatibility and to allow SQLite users to specify the isolation level when beginning transactions.

For SQLite users, the valid isolation levels for `lock_type` are:

- `exclusive`
- `immediate`
- `deferred`

Example usage:

```
# Calling transaction() in turn calls begin('exclusive').
with db.transaction('exclusive'):
    # No other readers or writers allowed while this is active.
    (Account
     .update(Account.balance=Account.balance - 100)
     .where(Account.id == from_acct)
     .execute())

    (Account
     .update(Account.balance=Account.balance + 100)
     .where(Account.id == to_acct)
     .execute())
```

commit()
Call `commit()` on the active connection, committing the current transaction.

rollback()
Call `rollback()` on the active connection, rolling back the current transaction.

set_autocommit(*autocommit*)

Parameters **autocommit** – a boolean value indicating whether to turn on/off autocommit.

get_autocommit()

Return type a boolean value indicating whether autocommit is enabled.

get_tables([*schema=None*])

Return type a list of table names in the database.

get_indexes(*table*[, *schema=None*])

Return type a list of *IndexMetadata* instances, representing the indexes for the given table.

get_columns(*table*[, *schema=None*])

Return type a list of *ColumnMetadata* instances, representing the columns for the given table.

get_primary_keys(*table*[, *schema=None*])

Return type a list containing the primary key column name(s) for the given table.

get_foreign_keys(*table*[, *schema=None*])

Return type a list of *ForeignKeyMetadata* instances, representing the foreign keys for the given table.

sequence_exists(*sequence_name*)

Rtype boolean

create_table(*model_class*[, *safe=True*])

Parameters

- **model_class** – *Model* class.
- **safe** (*bool*) – If *True*, the table will not be created if it already exists.

> **Warning:** Unlike *Model.create_table()*, this method does not create indexes or constraints. This method will only create the table itself. If you wish to create the table along with any indexes and constraints, use either *Model.create_table()* or *Database.create_tables()*.

create_index(*model_class*, *fields*[, *unique=False*])

Parameters

- **model_class** – *Model* table on which to create index
- **fields** – field(s) to create index on (either field instances or field names)
- **unique** – whether the index should enforce uniqueness

create_foreign_key(*model_class*, *field*[, *constraint=None*])

Parameters

- **model_class** – *Model* table on which to create foreign key constraint
- **field** – `Field` object
- **constraint** (*str*) – Name to give foreign key constraint.

Manually create a foreign key constraint using an `ALTER TABLE` query. This is primarily used when creating a circular foreign key dependency, for example:

```
DeferredPost = DeferredRelation()

class User(Model):
    username = CharField()
    favorite_post = ForeignKeyField(DeferredPost, null=True)

class Post(Model):
    title = CharField()
    author = ForeignKeyField(User, related_name='posts')

DeferredPost.set_model(Post)

# Create tables.  The foreign key from Post -> User will be created
# automatically, but the foreign key from User -> Post must be added
# manually.
User.create_table()
Post.create_table()

# Manually add the foreign key constraint on `User`, since we could
# not add it until we had created the `Post` table.
db.create_foreign_key(User, User.favorite_post)
```

create_sequence(*sequence_name*)

Parameters **sequence_name** – name of sequence to create

> **Note:** only works with database engines that support sequences

drop_table(*model_class*[, *fail_silently=False*[, *cascade=False*]])

Parameters

- **model_class** – *Model* table to drop

- **fail_silently** (*bool*) – if `True`, query will add a `IF EXISTS` clause
- **cascade** (*bool*) – drop table with `CASCADE` option.

drop_sequence(*sequence_name*)

Parameters **sequence_name** – name of sequence to drop

Note: only works with database engines that support sequences

---

create_tables(*models*[, *safe=False*])

Parameters

- **models** (*list*) – A list of models.
- **safe** (*bool*) – Check first whether the table exists before attempting to create it.

This method should be used for creating tables as it will resolve the model dependency graph and ensure the tables are created in the correct order. This method will also create any indexes and constraints defined on the models.

Usage:

```
db.create_tables([User, Tweet, Something], safe=True)
```

drop_tables(*models*[, *safe=False*[, *cascade=False*]])

Parameters

- **models** (*list*) – A list of models.
- **safe** (*bool*) – Check the table exists before attempting to drop it.
- **cascade** (*bool*) – drop table with `CASCADE` option.

This method should be used for dropping tables, as it will resolve the model dependency graph and ensure the tables are dropped in the correct order.

Usage:

```
db.drop_tables([User, Tweet, Something], safe=True)
```

atomic([*transaction_type=None*])

Execute statements in either a transaction or a savepoint. The outer-most call to *atomic* will use a transaction, and any subsequent nested calls will use savepoints.

Parameters **transaction_type** (*str*) – Specify isolation level. This parameter only has effect on **SQLite databases**, and furthermore, only affects the outer-most call to *atomic()*. For more information, see *transaction()*.

`atomic` can be used as either a context manager or a decorator.

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

Context manager example code:

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

    with db.atomic() as nested_txn:
        do_other_things()
        if something_bad_happened():
            # Roll back these changes, but preserve the changes
            # made in the outer block.
            nested_txn.rollback()
```

Decorator example code:

```
@db.atomic()
def create_user(username):
    # This function will execute in a transaction/savepoint.
    return User.create(username=username)
```

transaction([*transaction_type=None*])

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

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

Parameters **transaction_type** (*str*) – Specify isolation level, **SQLite only**.

Context manager example code:

```
# delete a blog instance and all its associated entries, but
# do so within a transaction
with database.transaction():
    blog.delete_instance(recursive=True)


# Explicitly roll back a transaction.
with database.transaction() as txn:
    do_some_stuff()
    if something_bad_happened():
        # Roll back any changes made within this block.
        txn.rollback()
```

Decorator example code:

```
@database.transaction()
def transfer_money(from_acct, to_acct, amt):
    from_acct.charge(amt)
    to_acct.pay(amt)
    return amt
```

SQLite users can specify the isolation level by specifying one of the following values for `transaction_type`:

- `exclusive`
- `immediate`
- `deferred`

Example usage:

```
with db.transaction('exclusive'):
    # No other readers or writers allowed while this is active.
    (Account
     .update(Account.balance=Account.balance - 100)
     .where(Account.id == from_acct)
     .execute())

    (Account
     .update(Account.balance=Account.balance + 100)
     .where(Account.id == to_acct)
     .execute())
```

commit_on_success (*func*)

Note: Use *atomic()* or *transaction()* instead.

savepoint ([*sid=None*])

Execute statements in a savepoint using either a context manager or decorator. If an error is raised inside the wrapped block, the savepoint will be rolled back, otherwise statements are committed when exiting. Like *transaction()*, a savepoint can also be explicitly rolled-back or committed by calling *rollback()* or *commit()*. If you manually commit or roll back, a new savepoint **will not** be created.

Savepoints can be thought of as nested transactions.

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

Context manager example code:

```
with db.transaction() as txn:
    do_some_stuff()
    with db.savepoint() as sp1:
        do_more_things()

    with db.savepoint() as sp2:
        even_more()
        # Oops, something bad happened, roll back
        # just the changes made in this block.
        if something_bad_happened():
            sp2.rollback()
```

execution_context ([*with_transaction=True*])

Create an *ExecutionContext* context manager or decorator. Blocks wrapped with an *ExecutionContext* will run using their own connection. By default, the wrapped block will also run in a transaction, although this can be disabled specifyin `with_transaction=False`.

For more explanation of *ExecutionContext*, see the *Advanced Connection Management* section.

Warning: ExecutionContext is very new and has not been tested extensively.

classmethod register_fields (*fields*)

Register a mapping of field overrides for the database class. Used to register custom fields or override the defaults.

Parameters **fields** (*dict*) – A mapping of db_field to column type

classmethod register_ops(*ops*)
Register a mapping of operations understood by the QueryCompiler to their SQL equivalent, e.g. {OP.EQ: '='}. Used to extend the types of field comparisons.

Parameters **fields** (*dict*) – A mapping of db_field to column type

extract_date(*date_part*, *date_field*)
Return an expression suitable for extracting a date part from a date field. For instance, extract the year from a *DateTimeField*.

Parameters

- **date_part** (*str*) – The date part attribute to retrieve. Valid options are: "year", "month", "day", "hour", "minute" and "second".
- **date_field** (*Field*) – field instance storing a datetime, date or time.

Return type an expression object.

truncate_date(*date_part*, *date_field*)
Return an expression suitable for truncating a date / datetime to the given resolution. This can be used, for example, to group a collection of timestamps by day.

Parameters

- **date_part** (*str*) – The date part to truncate to. Valid options are: "year", "month", "day", "hour", "minute" and "second".
- **date_field** (*Field*) – field instance storing a datetime, date or time.

Return type an expression object.

Example:

```
# Get tweets from today.
tweets = Tweet.select().where(
    db.truncate_date('day', Tweet.timestamp) == datetime.date.today())
```

class SqliteDatabase(*Database*)
Database subclass that works with the sqlite3 driver (or pysqlite2). In addition to the default database parameters, *SqliteDatabase* also accepts a *journal_mode* parameter which will configure the journaling mode.

Note: If you have both sqlite3 and pysqlite2 installed on your system, peewee will use whichever points at a newer version of SQLite.

Note: SQLite is unique among the databases supported by Peewee in that it allows a high degree of customization by the host application. This means you can do things like write custom functions or aggregates *in Python* and then call them from your SQL queries. This feature, and many more, are available through the *SqliteExtDatabase*, part of playhouse.sqlite_ext. I *strongly* recommend you use *SqliteExtDatabase* as it exposes many of the features that make SQLite so powerful.

Custom parameters:

Parameters

- **journal_mode** (*str*) – Journaling mode.

- **pragmas** (*list*) – List of 2-tuples containing PRAGMA statements to run against new connections.

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

```
db = SqliteDatabase('my_app.db', pragmas=(
    ('journal_mode', 'WAL'),
    ('cache_size', 10000),
    ('mmap_size', 1024 * 1024 * 32),
))
```

insert_many = True *if* using SQLite 3.7.11.0 or newer.

class MySQLDatabase(*Database*)

Database subclass that works with either "MySQLdb" or "pymysql".

commit_select = True

compound_operations = ['UNION']

for_update = True

subquery_delete_same_table = False

class PostgresqlDatabase(*Database*)

Database subclass that works with the "psycopg2" driver

commit_select = True

compound_select_parentheses = True

distinct_on = True

for_update = True

for_update_nowait = True

insert_returning = True

returning_clause = True

sequences = True

window_functions = True

register_unicode = True

Control whether the UNICODE and UNICODEARRAY psycopg2 extensions are loaded automatically.

Transaction, Savepoint and ExecutionContext

The easiest way to create transactions and savepoints is to use *Database.atomic()*. The *atomic()* method will create a transaction or savepoint depending on the level of nesting.

```
with db.atomic() as txn:
    # The outer-most call will be a transaction.
    with db.atomic() as sp:
        # Nested calls will be savepoints instead.
        execute_some_statements()
```

class transaction(*database*)

Context manager that encapsulates a database transaction. Statements executed within the wrapped block will be committed at the end of the block unless an exception occurs, in which case any changes will be rolled back.

> **Warning:** Transactions should not be nested as this could lead to unpredictable behavior in the event of an exception in a nested block. If you wish to use nested transactions, use the *atomic()* method, which will create a transaction at the outer-most layer and use savepoints for nested blocks.

Note: In practice you should not create *transaction* objects directly, but rather use the *Database.transaction()* method.

---

commit()

Manually commit any pending changes and begin a new transaction.

rollback()

Manually roll-back any pending changes and begin a new transaction.

class savepoint(*database*[, *sid=None*])

Context manager that encapsulates a savepoint (nested transaction). Statements executed within the wrapped block will be committed at the end of the block unless an exception occurs, in which case any changes will be rolled back.

> **Warning:** Savepoints must be created within a transaction. It is recommended that you use *atomic()* instead of manually managing the transaction+savepoint stack.

Note: In practice you should not create *savepoint* objects directly, but rather use the *Database.savepoint()* method.

---

commit()

Manually commit any pending changes. If the savepoint is manually committed and additional changes are made, they will be executed in the context of the outer block.

rollback()

Manually roll-back any pending changes. If the savepoint is manually rolled-back and additional changes are made, they will be executed in the context of the outer block.

class ExecutionContext(*database*[, *with_transaction=True*])

ExecutionContext provides a way to explicitly run statements in a dedicated connection. Typically a single database connection is maintained per-thread, but in some situations you may wish to explicitly force a new, separate connection. To accomplish this, you can create an *ExecutionContext*. Statements executed in the wrapped block will be run in a transaction by default, though you can disable this by specifying `with_transaction=False`.

Note: Rather than instantiating `ExecutionContext` directly, use *Database.execution_context()*.

Example code:

```
# This will return the connection associated with the current thread.
conn = db.get_conn()

with db.execution_context():
    # This will be a new connection object. If you are using the
    # connection pool, it may be an unused connection from the pool.
    ctx_conn = db.get_conn()

    # This statement is executed using the new `ctx_conn`.
    User.create(username='huey')

# At the end of the wrapped block, the connection will be closed and the
# transaction, if one exists, will be committed.

# This statement is executed using the regular `conn`.
User.create(username='mickey')
```

class `Using`(*database*, *models*[, *with_transaction=True*])

For the duration of the wrapped block, all queries against the given `models` will use the specified `database`. Optionally these queries can be run outside a transaction by specifying `with_transaction=False`.

`Using` provides, in short, a way to run queries on a list of models using a manually specified database.

Parameters

- **database** – a *Database* instance.
- **models** – a list of *Model* classes to use with the given database.
- **with_transaction** – Whether the wrapped block should be run in a transaction.

Warning: The *Using* context manager does not do anything to manage the database connections, so it the user's responsibility to make sure that you close the database explicitly.

Example:

```
master = PostgresqlDatabase('master')
replica = PostgresqlDatabase('replica')

class Data(Model):
    value = IntegerField()
    class Meta:
        database = master

# All these queries use the "master" database,
# since that is what our Data model was configured
# to use.
for i in range(10):
    Data.create(value=i)

Data.insert_many({Data.value: j} for j in range(100, 200)).execute()

# To use the read replica, we can use the Using context manager.
with Using(read_replica, [Data]):
    # Query is executed against the read replica.
    n_data = Data.select().count()

    # Since we did not specify this model in the list passed
```

```
    # to Using, it will use whatever database it was defined with.
    other_count = SomeOtherModel.select().count()
```

Metadata Types

class **IndexMetadata**(*name*, *sql*, *columns*, *unique*, *table*)

name
The name of the index.

sql
The SQL query used to generate the index.

columns
A list of columns that are covered by the index.

unique
A boolean value indicating whether the index has a unique constraint.

table
The name of the table containing this index.

class **ColumnMetadata**(*name*, *data_type*, *null*, *primary_key*, *table*)

name
The name of the column.

data_type
The data type of the column

null
A boolean value indicating whether `NULL` is permitted in this column.

primary_key
A boolean value indicating whether this column is a primary key.

table
The name of the table containing this column.

class **ForeignKeyMetadata**(*column*, *dest_table*, *dest_column*, *table*)

column
The column containing the foreign key (the "source").

dest_table
The table referenced by the foreign key.

dest_column
The column referenced by the foreign key (on `dest_table`).

table
The name of the table containing this foreign key.

Misc

class fn

A helper class that will convert arbitrary function calls to SQL function calls.

To express functions in peewee, use the *fn* object. The way it works is anything to the right of the "dot" operator will be treated as a function. You can pass that function arbitrary parameters which can be other valid expressions.

For example:

Peewee expression	Equivalent SQL
`fn.Count(Tweet.id).alias('count')`	`Count(t1."id") AS count`
`fn.Lower(fn.Substr(User.username, 1, 1))`	`Lower(Substr(t1."username", 1, 1))`
`fn.Rand().alias('random')`	`Rand() AS random`
`fn.Stddev(Employee.salary).alias('sdv')`	`Stddev(t1."salary") AS sdv`

over(*[partition_by=None[, order_by=None[, start=None[, end=None[, window=None]]]]]*)

Basic support for SQL window functions.

Parameters

- **partition_by** (*list*) – List of *Node* instances to partition by.
- **order_by** (*list*) – List of *Node* instances to use for ordering.
- **start** – The start of the *frame* of the window query.
- **end** – The end of the *frame* of the window query.
- **window** (Window) – A *Window* instance to use for this aggregate.

Examples:

```
# Get the list of employees and the average salary for their dept.
query = (Employee
         .select(
             Employee.name,
             Employee.department,
             Employee.salary,
             fn.Avg(Employee.salary).over(
                 partition_by=[Employee.department]))
         .order_by(Employee.name))

# Rank employees by salary.
query = (Employee
         .select(
             Employee.name,
             Employee.salary,
             fn.rank().over(
                 order_by=[Employee.salary])))

# Get a list of page-views, along with avg pageviews for that day.
query = (PageView
         .select(
             PageView.url,
             PageView.timestamp,
             fn.Count(PageView.id).over(
                 partition_by=[fn.date_trunc(
```

```
                        'day', PageView.timestamp)]))
          .order_by(PageView.timestamp))

# Same as above but using a window class.
window = Window(partition_by=[fn.date_trunc('day', PageView.timestamp)])
query = (PageView
          .select(
              PageView.url,
              PageView.timestamp,
              fn.Count(PageView.id).over(window=window))
          .window(window)  # Need to include our Window here.
          .order_by(PageView.timestamp))

# Get the list of times along with the last time.
query = (Times
          .select(
               Times.time,
               fn.LAST_VALUE(Times.time).over(
                   order_by=[Times.time],
                   start=Window.preceding(),
                   end=Window.following())))
```

class **SQL** (*sql*, **params*)

Add fragments of SQL to a peewee query. For example you might want to reference an aliased name.

Parameters

- **sql** (*str*) – Arbitrary SQL string.
- **params** – Arbitrary query parameters.

```
# Retrieve user table and "annotate" it with a count of tweets for each
# user.
query = (User
          .select(User, fn.Count(Tweet.id).alias('ct'))
          .join(Tweet, JOIN.LEFT_OUTER)
          .group_by(User))

# Sort the users by number of tweets.
query = query.order_by(SQL('ct DESC'))
```

class **Window** ([*partition_by=None*[, *order_by=None*[, *start=None*[, *end=None*]]]])

Create a `WINDOW` definition.

Parameters

- **partition_by** (*list*) – List of *Node* instances to partition by.
- **order_by** (*list*) – List of *Node* instances to use for ordering.
- **start** – The start of the *frame* of the window query.
- **end** – The end of the *frame* of the window query.

Examples:

```
# Get the list of employees and the average salary for their dept.
window = Window(partition_by=[Employee.department]).alias('dept_w')
query = (Employee
          .select(
              Employee.name,
```

```
            Employee.department,
            Employee.salary,
            fn.Avg(Employee.salary).over(window))
        .window(window)
        .order_by(Employee.name))
```

static preceding([*value=None*])
Return an expression appropriate for passing in to the `start` or `end` clause of a *Window* object. If `value` is not provided, then it will be `UNBOUNDED PRECEDING`.

static following([*value=None*])
Return an expression appropriate for passing in to the `start` or `end` clause of a *Window* object. If `value` is not provided, then it will be `UNBOUNDED FOLLOWING`.

class DeferredRelation
Used to reference a not-yet-created model class. Stands in as a placeholder for the related model of a foreign key. Useful for circular references.

```
DeferredPost = DeferredRelation()

class User(Model):
    username = CharField()

    # `Post` is not available yet, it is declared below.
    favorite_post = ForeignKeyField(DeferredPost, null=True)

class Post(Model):
    # `Post` comes after `User` since it refers to `User`.
    user = ForeignKeyField(User)
    title = CharField()

DeferredPost.set_model(Post)  # Post is now available.
```

set_model(*model*)
Replace the placeholder with the correct model class.

class Proxy
Proxy class useful for situations when you wish to defer the initialization of an object. For instance, you want to define your models but you do not know what database engine you will be using until runtime.

Example:

```
database_proxy = Proxy()  # 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)
```

initialize(*obj*)

Parameters **obj** – The object to proxy to.

Once initialized, the attributes and methods on `obj` can be accessed directly via the *Proxy* instance.

class Node

The *Node* class is the parent class for all composable parts of a query, and forms the basis of peewee's expression API. The following classes extend *Node*:

- *SelectQuery*, *UpdateQuery*, *InsertQuery*, *DeleteQuery*, and *RawQuery*.
- `Field`
- `Func` (and *fn()*)
- *SQL*
- `Expression`
- `Param`
- *Window*
- `Clause`
- `Entity`
- `Check`

Overridden operators:

- Bitwise and- and or- (`&` and `|`): combine multiple nodes using the given conjunction.
- `+`, `-`, `*`, `/` and `^` (add, subtract, multiply, divide and exclusive-or).
- `==`, `!=`, `<`, `<=`, `>`, `>=`: create a binary expression using the given comparator.
- `<<`: create an *IN* expression.
- `>>`: create an *IS* expression.
- `%` and `**`: *LIKE* and *ILIKE*.

contains(*rhs*)
Create a binary expression using case-insensitive string search.

startswith(*rhs*)
Create a binary expression using case-insensitive prefix search.

endswith(*rhs*)
Create a binary expression using case-insensitive suffix search.

between(*low*, *high*)
Create an expression that will match values between `low` and `high`.

regexp(*expression*)
Match based on regular expression.

concat(*rhs*)
Concatenate the current node with the provided `rhs`.

> **Warning:** In order for this method to work with MySQL, the MySQL session must be set to use `PIPES_AS_CONCAT`.
>
> To reliably concatenate strings with MySQL, use `fn.CONCAT(s1, s2...)` instead.

is_null([*is_null=True*])
Create an expression testing whether the `Node` is (or is not) `NULL`.

```
# Find all categories whose parent column is NULL.
root_nodes = Category.select().where(Category.parent.is_null())

# Find all categores whose parent is NOT NULL.
child_nodes = Category.select().where(Category.parent.is_null(False))
```

To simplify things, peewee will generate the correct SQL for equality and inequality. The *is_null()* method is provided simply for readability.

```
# Equivalent to the previous queries -- peewee will translate these
# into `IS NULL` and `IS NOT NULL`:
root_nodes = Category.select().where(Category.parent == None)
child_nodes = Category.select().where(Category.parent != None)
```

__invert__()
Negate the node. This translates roughly into *NOT (<node>)*.

alias([*name=None*])
Apply an alias to the given node. This translates into *<node> AS <name>*.

asc()
Apply ascending ordering to the given node. This translates into *<node> ASC*.

desc()
Apply descending ordering to the given node. This translates into *<node> DESC*.

bind_to(*model_class*)
Bind the results of an expression to a specific model type. Useful when adding expressions to a select, where the result of the expression should be placed on a particular joined instance.

classmethod extend([*name=None*[, *clone=False*]])
Decorator for adding the decorated function as a new method on *Node* and its subclasses. Useful for adding implementation-specific features to all node types.

Parameters

- **name** (*str*) – Method name. If not provided the name of the wrapped function will be used.
- **clone** (*bool*) – Whether this method should return a clone. This is generally true when the method mutates the internal state of the node.

Example:

```
# Add a `cast()` method to all nodes using the '::' operator.
PostgresqlDatabase.register_ops({'::', '::'})

@Node.extend()
def cast(self, as_type):
    return Expression(self, '::', SQL(as_type))
```

```
# Let's pretend we want to find all data points whose numbers
# are palindromes. Note that we can use the new *cast* method
# on both fields and with the `fn` helper:
reverse_val = fn.REVERSE(DataModel.value.cast('str')).cast('int')

query = (DataPoint
         .select()
         .where(DataPoint.value == reverse_val))
```

Note: To remove an extended method, simply call `delattr` on the class the method was originally added to.

Hacks

Collected hacks using peewee. Have a cool hack you'd like to share? Open an issue on GitHub or contact me.

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 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)
        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.
            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
```

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 create_date for each
# user.
subquery = (TweetAlias
            .select(
                TweetAlias.user,
                fn.MAX(TweetAlias.create_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 create_date.
query = (Tweet
         .select(Tweet, User)
         .join(User)
         .switch(Tweet)
         .join(subquery, on=(
             (Tweet.create_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.create_date == fn.MAX(Tweet.create_date)))
```

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 t2.id, t2.username FROM user AS t2) AS uq
   LEFT JOIN LATERAL
  (SELECT t2.message, t2.create_date
   FROM tweet AS t2
   WHERE (t2.user_id = uq.id)
   ORDER BY t2.create_date DESC LIMIT 3)
  AS pq ON true
```

To accomplish this with peewee we'll need to express the lateral join as a `Clause`, which gives us greater flexibility than the *`join()`* method.

```
# We'll reference `Tweet` twice, so keep an alias handy.
TweetAlias = Tweet.alias()

# The "outer loop" will be iterating over the users whose
# tweets we are trying to find.
user_query = User.select(User.id, User.username).alias('uq')

# The inner loop will select tweets and is correlated to the
# outer loop via the WHERE clause. Note that we are using a
# LIMIT clause.
tweet_query = (TweetAlias
               .select(TweetAlias.message, TweetAlias.create_date)
               .where(TweetAlias.user == user_query.c.id)
               .order_by(TweetAlias.create_date.desc())
               .limit(3)
               .alias('pq'))

# Now we join the outer and inner queries using the LEFT LATERAL
# JOIN. The join predicate is *ON TRUE*, since we're effectively
# joining in the tweet subquery's WHERE clause.
join_clause = Clause(
    user_query,
    SQL('LEFT JOIN LATERAL'),
    tweet_query,
    SQL('ON %s', True))

# Finally, we'll wrap these up and SELECT from the result.
query = (Tweet
         .select(SQL('*'))
         .from_(join_clause))
```

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
            ORDER BY t2.create_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.create_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.create_date >= Tweet.create_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.create_date >= Tweet.create_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.create_date.desc())
             .limit(3))))
```

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)
    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
```

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

N

O

P

Q

R

S

T

U

V

W

Y