peewee Documentation

Release 1.0.0

charles leifer

January 12, 2013

CONTENTS

- a small orm
- written in python
- provides a lightweight querying interface over sql
- uses sql concepts when querying, like joins and where clauses
- support for special extensions like hstore and full-text search

For flask integration, including an admin interface and RESTful API, check out flask-peewee.

CHAPTER

ONE

CONTENTS:

1.1 Overview

peewee is a lightweight ORM written in python.

Examples:

```
# a simple query selecting a user
User.get(username='charles')

# get the staff and super users
editors = User.select().where(Q(is_staff=True) | Q(is_superuser=True))

# get tweets by editors
Tweet.select().where(user__in=editors)

# how many active users are there?
User.select().where(active=True).count()

# paginate the user table and show me page 3 (users 41-60)
User.select().order_by(('username', 'asc')).paginate(3, 20)

# order users by number of tweets
User.select().annotate(Tweet).order_by(('count', 'desc'))

# another way of expressing the same
User.select({
    User: ['*'],
    Tweet: [Count('id', 'count')]
}).group_by('id').join(Tweet).order_by(('count', 'desc'))

# do an atomic update
TweetCount.update(count=F('count') + 1).where(user=charlie)
```

You can use django-style syntax to create select queries:

```
# how many active users are there?
User.filter(active=True).count()

# get tweets by a specific user
Tweet.filter(user__username='charlie')

# get tweets by editors
Tweet.filter(Q(user__is_staff=True) | Q(user__is_superuser=True))
```

You can use python operators to create select queries:

```
# how many active users are there?
User.select().where(User.active == True).count()

# get me all users in their thirties
User.select().where((User.age >= 30) & (User.age < 40))

# get me tweets from today by active users
Tweet.select().join(User).where(
    (Tweet.pub_date >= today) &
    (User.active == True)
)
```

Check out *the docs* for notes on the methods of querying.

1.1.1 Why?

peewee began when I was working on a small app in flask and found myself writing lots of queries and wanting a very simple abstraction on top of the sql. I had so much fun working on it that I kept adding features. My goal has always been, though, to keep the implementation incredibly simple. I've made a couple dives into django's orm but have never come away with a deep understanding of its implementation. peewee is small enough that its my hope anyone with an interest in orms will be able to understand the code without too much trouble.

1.2 Installing peewee

```
pip install peewee
```

1.2.1 Installing with git

You can pip install the git clone:

```
pip install -e git+https://github.com/coleifer/peewee.git
```

If you don't want to use pip:

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

You can test your installation by running the test suite.

```
python setup.py test
```

Feel free to check out the *Example app* which ships with the project.

1.3 Peewee Cookbook

Below are outlined some of the ways to perform typical database-related tasks with peewee.

Examples will use the following models:

```
import peewee

class Blog(peewee.Model):
    creator = peewee.CharField()
    name = peewee.CharField()


class Entry(peewee.Model):
    blog = peewee.ForeignKeyField(Blog)
    title = peewee.CharField()
    body = peewee.TextField()
    pub_date = peewee.DateTimeField()
    published = peewee.BooleanField(default=True)
```

1.3.1 Database and Connection Recipes

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.

```
>>> database = SqliteDatabase('stats.db')
>>> database.connect()
```

To use this database with your models, specify it in an inner "Meta" class:

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

    class Meta:
        database = database
```

It is possible to use multiple databases (provided that you don't try and mix models from each):

```
>>> custom_db = peewee.SqliteDatabase('custom.db')

>>> class CustomModel(peewee.Model):
...     whatev = peewee.CharField()
...
...     class Meta:
...         database = custom_db
...

>>> custom_db.connect()
>>> CustomModel.create_table()
```

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:

```
custom_db = peewee.SqliteDatabase('custom.db')

class CustomModel(peewee.Model):
    class Meta:
        database = custom_db

class Blog(CustomModel):
    creator = peewee.CharField()
    name = peewee.TextField()

class Entry(CustomModel):
    # etc, etc
```

Note: Remember to specify a database in a model class (or its parent class), otherwise peewee will fall back to a default sqlite database named "peewee.db".

Using with Postgresql

Point models at an instance of `PostgresqlDatabase`.

```
psql_db = peewee.PostgresqlDatabase('my_database', user='code')


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

class Blog(PostgresqlModel):
    creator = peewee.CharField()
    # etc, etc
```

Using with MySQL

Point models at an instance of `MySQLDatabase`.

```
mysql_db = peewee.MySQLDatabase('my_database', user='code')


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

class Blog(MySQLModel):
    creator = peewee.CharField()
    # etc, etc


# when you're ready to start querying, remember to connect
mysql_db.connect()
```

Multi-threaded applications

Some database engines may not allow a connection to be shared across threads, notably sqlite. If you would like peewee to maintain a single connection per-thread, instantiate your database with `threadlocals=True`:

```
concurrent_db = SqliteDatabase('stats.db', threadlocals=True)
```

The above implementation stores connection state in a thread local and will only use that connection for a given thread. Sqlite can also share a connection across threads natively, so if you would prefer to use sqlite's native support:

```
native_concurrent_db = SqliteDatabase('stats.db', check_same_thread=False)
```

Deferring initialization

Sometimes the database information is not known until run-time, when it might be loaded from a configuration file/etc. In this case, you can "defer" the initialization of the database by passing in `None` as the database_name.

```
deferred_db = peewee.SqliteDatabase(None)

class SomeModel(peewee.Model):
    class Meta:
        database = deferred_db
```

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

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

To initialize your database, you simply call the `init` method with the database_name and any additional kwargs:

```
database_name = raw_input('What is the name of the db? ')
deferred_db.init(database_name)
```

1.3.2 Creating, Reading, Updating and Deleting

Creating a new record

You can use the `Model.create()` method on the model:

```
>>> Blog.create(creator='Charlie', name='My Blog')
<__main__.Blog 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 save it:

```
>>> blog = Blog()
>>> blog.creator = 'Chuck'
>>> blog.name = 'Another blog'
>>> blog.save()
>>> blog.id
2
```

Updating existing records

Once a model instance has a primary key, any attempt to re-save it will result in an `UPDATE` rather than another `INSERT`:

```
>>> blog.save()
>>> blog.id
2
>>> blog.save()
>>> blog.id
2
```

If you want to update multiple records, issue an UPDATE query. The following example will update all Entry objects, marking them as "published", if their pub_date is less than today's date.

```
>>> update_query = Entry.update(published=True).where(pub_date__lt=datetime.today())
>>> update_query.execute()
4 # <--- number of rows updated
```

For more information, see the documentation on UpdateQuery.

Deleting a record

To delete a single model instance, you can use the Model.delete_instance() shortcut:

```
>>> blog = Blog.get(id=1)
>>> blog.delete_instance()
1 # <--- number of rows deleted

>>> Blog.get(id=1)
BlogDoesNotExist: instance matching query does not exist:
SQL: SELECT "id", "creator", "name" FROM "blog" WHERE "id" = ? LIMIT 1
PARAMS: [1]
```

To delete an arbitrary group of records, you can issue a DELETE query. The following will delete all Entry objects that are a year old.

```
>>> delete_query = Entry.delete().where(pub_date__lt=one_year_ago)
>>> delete_query.execute()
7 # <--- number of entries deleted
```

For more information, see the documentation on DeleteQuery.

Selecting a single record

You can use the Model.get() method to retrieve a single instance matching the given query (passed in as a mix of Q objects and keyword arguments).

This method is a shortcut that calls Model.select() with the given query, but limits the result set to 1. Additionally, if no model matches the given query, a DoesNotExist exception will be raised.

```
>>> Blog.get(id=1)
<__main__.Blog object at 0x25294d0>

>>> Blog.get(id=1).name
u'My Blog'

>>> Blog.get(creator='Chuck')
<__main__.Blog object at 0x2529410>

>>> Blog.get(id=1000)
BlogDoesNotExist: instance matching query does not exist:
SQL: SELECT "id", "creator", "name" FROM "blog" WHERE "id" = ? LIMIT 1
PARAMS: [1000]
```

For more information see notes on SelectQuery and *Querying API* in general.

Selecting multiple records

To simply get all instances in a table, call the `Model.select()` method:

```
>>> for blog in Blog.select():
...     print blog.name
...
My Blog
Another blog
```

When you iterate over a `SelectQuery`, it will automatically execute it and start returning results from the database cursor. Subsequent iterations of the same query will not hit the database as the results are cached.

Another useful note is that you can retrieve instances related by `ForeignKeyField` by iterating. To get all the related instances for an object, you can query the related name. Looking at the example models, we have Blogs and Entries. Entry has a foreign key to Blog, meaning that any given blog may have 0..n entries. A blog's related entries are exposed using a `SelectQuery`, and can be iterated the same as any other SelectQuery:

```
>>> for entry in blog.entry_set:
...     print entry.title
...
entry 1
entry 2
entry 3
entry 4
```

The `entry_set` attribute is just another select query and any methods available to `SelectQuery` are available:

```
>>> for entry in blog.entry_set.order_by(('pub_date', 'desc')):
...     print entry.title
...
entry 4
entry 3
entry 2
entry 1
```

Filtering records

```
>>> for entry in Entry.select().where(blog=blog, published=True):
...     print '%s: %s (%s)' % (entry.blog.name, entry.title, entry.published)
...
My Blog: Some Entry (True)
My Blog: Another Entry (True)

>>> for entry in Entry.select().where(pub_date__lt=datetime.datetime(2011, 1, 1)):
...     print entry.title, entry.pub_date
...
Old entry 2010-01-01 00:00:00
```

You can also filter across joins:

```
>>> for entry in Entry.select().join(Blog).where(name='My Blog'):
...     print entry.title
Old entry
Some Entry
Another Entry
```

If you are already familiar with Django's ORM, you can use the "double underscore" syntax:

```
>>> for entry in Entry.filter(blog__name='My Blog'):
...     print entry.title
Old entry
Some Entry
Another Entry
```

If you prefer, you can use python operators to query:

```
>>> for entry in Entry.select().join(Blog).where(Blog.name=='My Blog')
...     print entry.title
```

To perform OR lookups, use the special `Q` object. These work in both calls to `filter()` and `where()`:

```
>>> User.filter(Q(staff=True) | Q(superuser=True)) # get staff or superusers
```

To perform lookups against *another column* in a given row, use the `F` object:

```
>>> Employee.filter(salary__lt=F('desired_salary'))
```

Check *the docs* for more examples of querying.

Sorting records

```
>>> for e in Entry.select().order_by('pub_date'):
...     print e.pub_date
...
2010-01-01 00:00:00
2011-06-07 14:08:48
2011-06-07 14:12:57

>>> for e in Entry.select().order_by(peewee.desc('pub_date')):
...     print e.pub_date
...
2011-06-07 14:12:57
2011-06-07 14:08:48
2010-01-01 00:00:00
```

You can also order across joins. Assuming you want to order entries by the name of the blog, then by pubdate desc:

```
>>> qry = Entry.select().join(Blog).order_by(
...     (Blog, 'name'),
...     (Entry, 'pub_date', 'DESC'),
... )

>>> qry.sql()
('SELECT t1.* FROM entry AS t1 INNER JOIN blog AS t2 ON t1.blog_id = t2.id ORDER BY t2.name ASC, t1.
```

Paginating records

The paginate method makes it easy to grab a "page" or records – it takes two parameters, *page_number*, and *items_per_page*:

```
>>> for entry in Entry.select().order_by('id').paginate(2, 10):
...     print entry.title
...
entry 10
entry 11
entry 12
entry 13
entry 14
entry 15
entry 16
entry 17
entry 18
entry 19
```

Counting records

You can count the number of rows in any select query:

```
>>> Entry.select().count()
100
>>> Entry.select().where(id__gt=50).count()
50
```

Iterating over lots of rows

To limit the amount of memory used by peewee when iterating over a lot of rows (i.e. you may be dumping data to csv), use the `iterator()` method on the `QueryResultWrapper`. 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 1M stat objects to dump to csv
stats_qr = Stat.select().execute()

# our imaginary serializer class
serializer = CSVSerializer()

# loop over all the stats and serialize
for stat in stats_qr.iterator():
    serializer.serialize_object(stat)
```

For simple queries you can see further speed improvements by using the `SelectQuery.naive()` query method. See the documentation for details on this optimization.

```
stats_query = Stat.select().naive() # note we are calling "naive()"
stats_qr = stats_query.execute()

for stat in stats_qr.iterator():
    serializer.serialize_object(stat)
```

Performing atomic updates

Use the special `F` object to perform an atomic update:

```
>>> MessageCount.update(count=F('count') + 1).where(user=some_user)
```

Aggregating records

Suppose you have some blogs and want to get a list of them along with the count of entries in each. First I will show you the shortcut:

```
query = Blog.select().annotate(Entry)
```

This is equivalent to the following:

```
query = Blog.select({
    Blog: ['*'],
    Entry: [Count('id')],
}).group_by(Blog).join(Entry)
```

The resulting query will return Blog objects with all their normal attributes plus an additional attribute 'count' which will contain the number of entries. By default it uses an inner join if the foreign key is not nullable, which means blogs without entries won't appear in the list. To remedy this, manually specify the type of join to include blogs with 0 entries:

```
query = Blog.select().join(Entry, 'left outer').annotate(Entry)
```

You can also specify a custom aggregator:

```
query = Blog.select().annotate(Entry, peewee.Max('pub_date', 'max_pub_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:

```
>>> Tag.select().join(PhotoTag).join(Photo).group_by(Tag).having('count(*) > 5').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(*) > 5
```

Suppose we want to grab the associated count and store it on the tag:

```
>>> Tag.select({
...     Tag: ['*'],
...     Photo: [Count('id', 'count')]
... }).join(PhotoTag).join(Photo).group_by(Tag).having('count(*) > 5').sql()

SELECT t1."id", t1."name", COUNT(t3."id") AS count
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(*) > 5
```

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 `R` object to construct queries:

```
# select the users' id, username and the first letter of their username, lower-cased
query = User.select(['id', 'username', R('LOWER(SUBSTR(username, 1, 1))', 'first_letter')])

# now filter this list to include only users whose username begins with "a"
a_users = query.where(R('first_letter=%s', 'a'))

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

This same functionality could be easily exposed as part of the where clause, the only difference being that the first letter is not selected and therefore not an attribute of the model instance:

```
a_users = User.filter(R('LOWER(SUBSTR(username, 1, 1)) = %s', 'a'))
```

We can write subqueries as part of a `SelectQuery`, for example counting the number of entries on a blog:

```
entry_query = R('(SELECT COUNT(*) FROM entry WHERE entry.blog_id=blog.id)', 'entry_count')
blogs = Blog.select(['id', 'name', entry_query]).order_by(('entry_count', 'desc'))

for blog in blogs:
    print blog.title, blog.entry_count
```

It is also possible to use subqueries as part of a where clause, for example finding blogs that have no entries:

```
no_entry_query = R('NOT EXISTS (SELECT * FROM entry WHERE entry.blog_id=blog.id)')
blogs = Blog.filter(no_entry_query)

for blog in blogs:
    print blog.name, ' has no entries'
```

1.3.3 Working with transactions

Context manager

You can execute queries within a transaction using the `transaction` context manager, which will issue a commit if all goes well, or a rollback if an exception is raised:

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

with db.transaction():
    blog.delete_instance(recursive=True) # delete blog and associated entries
```

Decorator

Similar to the context manager, you can decorate functions with the `commit_on_success` decorator:

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

@db.commit_on_success
def delete_blog(blog):
    blog.delete_instance(recursive=True)
```

Changing autocommit behavior

By default, databases are initialized with `autocommit=True`, you can turn this on and off at runtime if you like. The behavior below is roughly the same as the context manager and decorator:

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

Blog.create(name='foo blog')
db.commit()
```

1.3.4 Introspecting databases

If you'd like to generate some models for an existing database, you can try out the database introspection tool "pwiz" that comes with peewee.

Usage:

```
python pwiz.py my_postgresql_database
```

It works with postgresql, mysql and sqlite:

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

pwiz will generate code for:

- database connection object
- a base model class to use this connection
- models that were introspected from the database tables

The generated code is written to stdout.

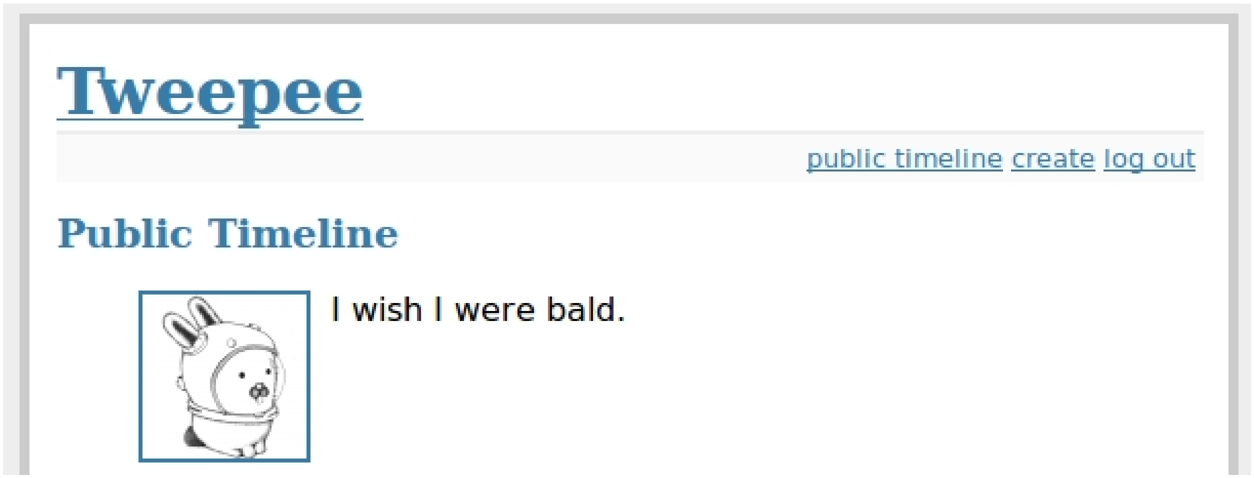
1.3.5 Schema migrations

Currently peewee does not have support for automatic schema migrations. Peewee does, however, come with a few helper functions:

- `Database.add_column_sql()`
- `Database.rename_column_sql()`
- `Database.drop_column_sql()`

Honestly, your best bet is to script any migrations and use plain ol' SQL.

1.4 Example app



Public Timeline

I wish I were bald.

peewee ships with an example web app that runs on the Flask microframework. If you already have flask and its dependencies installed you should be good to go, otherwise install from the included requirements file.

```
cd example/
pip install -r requirements.txt
```

1.4.1 Running the example

After ensuring that flask, jinja2, werkzeug and sqlite3 are all installed, switch to the example directory and execute the *run_example.py* script:

```
python run_example.py
```

1.4.2 Diving into the code

Models

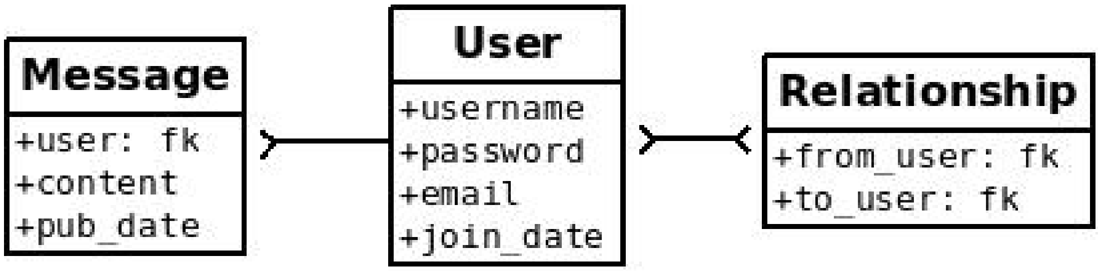
In the spirit of the ur-python framework, django, peewee uses declarative model definitions. If you're not familiar with django, the idea is that you declare a class with some members which map directly to the database schema. For the twitter clone, there are just three models:

`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 represents *"following"*.

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

If you like UML, this is basically what it looks like:



Here is what the code looks like:

```
database = SqliteDatabase(DATABASE)

# model definitions
class BaseModel(Model):
    class Meta:
        database = database

class User(BaseModel):
    username = CharField()
    password = CharField()
    email = CharField()
    join_date = DateTimeField()

    def following(self):
        return User.select().join(
            Relationship, on='to_user_id'
        ).where(from_user=self).order_by('username')

    def followers(self):
        return User.select().join(
            Relationship
        ).where(to_user=self).order_by('username')

    def is_following(self, user):
        return Relationship.select().where(
            from_user=self,
            to_user=user
        ).count() > 0

    def gravatar_url(self, size=80):
        return 'http://www.gravatar.com/avatar/%s?d=identicon&s=%d' % \
            (md5(self.email.strip().lower().encode('utf-8')).hexdigest(), size)


class Relationship(BaseModel):
    from_user = ForeignKeyField(User, related_name='relationships')
    to_user = ForeignKeyField(User, related_name='related_to')


class Message(BaseModel):
    user = ForeignKeyField(User)
    content = TextField()
    pub_date = DateTimeField()
```

peewee supports a handful of field types which map to different column types in sqlite. Conversion between python and the database is handled transparently, including the proper handling of `None`/`NULL`.

Note: You might have noticed that we created a `BaseModel` which sets the database, and then all the other models extend the `BaseModel`. This is a good way to make sure all your models are talking to the right database.

Creating the initial 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.

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

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

The `create_tables()` method is defined in the app module and looks like this:

```
def create_tables():
    User.create_table()
    Relationship.create_table()
    Message.create_table()
```

Every model has a `create_table()` classmethod which runs a `CREATE TABLE` statement in the database. Usually this is something you'll only do once, whenever a new model is added.

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 `ALTER TABLE`.

Note: If you want, you can use instead write `User.create_table(True)` and it will fail silently if the table already exists.

Connecting to the database

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, defaulting to "peewee.db". Since you probably want a bit more control, you can instantiate your own database and point your models at it. This is a peewee idiom:

```
# config
DATABASE = 'tweepee.db'

# ... more config here, omitted

database = SqliteDatabase(DATABASE) # tell our models to use "tweepee.db"
```

Because sqlite likes to have a separate connection per-thread, 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():
    g.db = database
    g.db.connect()
```

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

Note: We're storing the db on the magical variable `g` - that's a flask-ism and can be ignored as an implementation detail. The meat of this code is in the idea that we connect to our db every request and close that connection every response. Django does the exact same thing.

Doing queries

In the `User` model there are a few instance methods that encapsulate some user-specific functionality, i.e.

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

These methods are rather similar in their implementation but with one key difference:

```
def following(self):
    return User.select().join(
        Relationship, on='to_user_id'
    ).where(from_user=self).order_by('username')

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

Specifying the foreign key manually instructs peewee to join on the `to_user_id` field. The queries end up looking like:

```
# following:
SELECT t1.*
FROM user AS t1
INNER JOIN relationship AS t2
    ON t1.id = t2.to_user_id  # <-- joining on to_user_id
WHERE t2.from_user_id = ?
ORDER BY username ASC

# followers
SELECT t1.*
FROM user AS t1
INNER JOIN relationship AS t2
    ON t1.id = t2.from_user_id # <-- joining on from_user_id
WHERE t2.to_user_id = ?
ORDER BY username ASC
```

Creating new objects

So what happens when a new user wants to join the site? Looking at the business end of the `join()` view, we can that it does a quick check to see if the username is taken, and if not executes a `create()`.

```
try:
    user = User.get(username=request.form['username'])
    flash('That username is already taken')
except User.DoesNotExist:
    user = User.create(
        username=request.form['username'],
        password=md5(request.form['password']).hexdigest(),
        email=request.form['email'],
        join_date=datetime.datetime.now()
    )
```

Much like the `create()` method, all models come with a built-in method called `get_or_create()` which is used when one user follows another:

```
Relationship.get_or_create(
    from_user=session['user'], # <-- the logged-in user
    to_user=user, # <-- the user they want to follow
)
```

Doing 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, it is necessary to do a subquery:

```
# python code
qr = Message.select().where(user__in=some_user.following())
```

Results in the following SQL query:

```
SELECT *
FROM message
WHERE user_id IN (
    SELECT t1.id
    FROM user AS t1
    INNER JOIN relationship AS t2
        ON t1.id = t2.to_user_id
    WHERE t2.from_user_id = ?
    ORDER BY username ASC
)
```

peewee supports doing subqueries on any `ForeignKeyField` or `PrimaryKeyField`.

What else is of interest here?

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, **kwargs):
    try:
        return model.get(**kwargs)
    except model.DoesNotExist:
        abort(404)
```

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

1.5 Model API (smells like django)

Models and their fields map directly to database tables and columns. Consider the following:

```
from peewee import *

db = SqliteDatabase('test.db')

# create a base model class that our application's models will extend
class BaseModel(Model):
    class Meta:
        database = db


class Blog(BaseModel):
    name = CharField() # <-- VARCHAR


class Entry(BaseModel):
    headline = CharField()
    content = TextField() # <-- TEXT
    pub_date = DateTimeField() # <-- DATETIME
    blog = ForeignKeyField(Blog) # <-- INTEGER referencing the Blog table
```

This is a typical example of how to specify models with peewee. There are several things going on:

1. Create an instance of a `Database`

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

This establishes an object, db, which is used by the models to connect to and query the database. There can be multiple database instances per application, but, as I hope is obvious, ForeignKeyField related models must be on the same database.

2. Create a base model class which specifies our database

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

Model configuration is kept namespaced in a special class called Meta – this convention is borrowed from Django, which does the same thing. Meta configuration is passed on to subclasses, so this code basically allows all our project's models to connect to our database.

3. Declare a model or two

```
class Blog(BaseModel):
    name = CharField()
```

Model definition is pretty similar to django or sqlalchemy – you basically define a class which represents a single table in the database, then its attributes (which are subclasses of Field) represent columns.

Models provide methods for creating/reading/updating/deleting rows in the database.

1.5.1 Creating tables

In order to start using these models, its necessary to open a connection to the database and create the tables first:

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

# create the tables
Blog.create_table()
Entry.create_table()
```

Note: Strictly speaking, the explicit call to connect() is not necessary, but it is good practice to be explicit about when you are opening and closing connections.

1.5.2 Model instances

Assuming you've created the tables and connected to the database, you are now free to create models and execute queries.

Creating models in the interactive interpreter is a snap.

1. Use the Model.create() classmethod:

```
>>> blog = Blog.create(name='Funny pictures of animals blog')
>>> entry = Entry.create(
...     headline='maru the kitty',
...     content='http://www.youtube.com/watch?v=xdhLQCYQ-nQ',
...     pub_date=datetime.datetime.now(),
...     blog=blog
... )

>>> entry.blog.name
'Funny pictures of animals blog'
```

2. Build up the instance programmatically:

```
>>> blog = Blog()
>>> blog.name = 'Another sweet blog'
>>> blog.save()
```

Traversing foriegn keys

As you can see from above, the foreign key from `Entry` to `Blog` can be traversed automatically:

```
>>> entry.blog.name
'Funny pictures of animals blog'
```

The reverse is also true, we can iterate a `Blog` objects associated `Entries`:

```
>>> for entry in blog.entry_set:
...     print entry.headline
...
maru the kitty
```

Under the hood, the `entry_set` attribute is just a `SelectQuery`:

```
>>> blog.entry_set
<peewee.SelectQuery object at 0x151f510>

>>> blog.entry_set.sql()
('SELECT * FROM entry WHERE blog_id = ?', [1])
```

1.5.3 Model options

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

```
from peewee import *

custom_db = SqliteDatabase('custom.db')

class CustomModel(Model):
    class Meta:
        database = custom_db
```

This instructs peewee that whenever a query is executed on `CustomModel` to use the custom 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.

There are several options you can specify as `Meta` attributes:

- database: specifies a `Database` instance to use with this model
- db_table: the name of the database table this model maps to

- indexes: a list of fields to index
- ordering: a sequence of columns to use as the default ordering for this model
- **pk_sequence: name of sequence to create for the primary key (peewee will autogenerate one** if not provided and the backend supports sequences).

Specifying indexes:

```
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),
        )
```

Example of ordering:

```
class Entry(Model):
    title = CharField()
    body = TextField()
    created = DateTimeField()

    class Meta:
        # order by created date descending, then title ascending
        ordering = (('created', 'desc'), 'title')
```

Note: These options are "inheritable", which means that you can define a database adapter on one model, then subclass that model and the child models will use that database.

```
my_db = PostgresqlDatabase('my_db')

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

class SomeModel(BaseModel):
    field1 = CharField()

    class Meta:
        ordering = ('field1',)
        # no need to define database again since it will be inherited from
        # the BaseModel
```

1.5.4 Model methods

class Model

save ([*force_insert=False*])

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:

```
>>> some_obj.title = 'new title' # <-- does not touch the database
>>> some_obj.save() # <-- change is persisted to the db
```

classmethod create (***attributes*)

Parameters attributes – key/value pairs of model attributes

Create an instance of the `Model` with the given attributes set.

example:

```
>>> user = User.create(username='admin', password='test')
```

delete_instance ([*recursive=False*])

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).

example:

```
>>> some_obj.delete_instance() # <-- it is gone forever
```

classmethod filter (**args*, ***kwargs*)

Parameters

- **args** – a list of `Q` or `Node` objects
- **kwargs** – a mapping of column + lookup to value, e.g. "age__gt=55"

Return type `SelectQuery` with appropriate `WHERE` clauses

Provides a django-like syntax for building a query. The key difference between `filter()` and `SelectQuery.where()` is that `filter()` supports traversing joins using django's "double-underscore" syntax:

```
>>> sq = Entry.filter(blog__title='Some Blog')
```

This method is chainable:

```
>>> base_q = User.filter(active=True)
>>> some_user = base_q.filter(username='charlie')
```

classmethod get (**args*, ***kwargs*)

Parameters

- **args** – a list of `Q` or `Node` objects
- **kwargs** – a mapping of column + lookup to value, e.g. "age__gt=55"

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(username=username, password=password)
```

This method is also expose via the `SelectQuery`:

```
>>> active = User.select().where(active=True)
>>> try:
...     user = active.get(username=username, password=password)
... except User.DoesNotExist:
...     user = None
```

classmethod get_or_create(***attributes*)

Parameters attributes – key/value pairs of model attributes

Return type a `Model` instance

Get the instance with the given attributes set. If the instance does not exist it will be created.

example:

```
>>> CachedObj.get_or_create(key=key, val=some_val)
```

classmethod select(*query=None*)

Return type a `SelectQuery` for the given `Model`

example:

```
>>> User.select().where(active=True).order_by('username')
```

classmethod update(***query*)

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

example:

```
>>> q = User.update(active=False).where(registration_expired=True)
>>> q.sql()
('UPDATE user SET active=? WHERE registration_expired = ?', [0, 1])
>>> q.execute() # <-- execute it
```

classmethod delete(***query*)

Return type a `DeleteQuery` for the given `Model`

example:

```
>>> q = User.delete().where(active=False)
>>> q.sql()
('DELETE FROM user WHERE active = ?', [0])
>>> q.execute() # <-- execute it
```

Warning: Assume you have a model instance – calling `model_instance.delete()` does **not** delete it.

classmethod insert(***query*)

Return type an `InsertQuery` for the given `Model`

example:

```
>>> q = User.insert(username='admin', active=True, registration_expired=False)
>>> q.sql()
('INSERT INTO user (username,active,registration_expired) VALUES (?,?,?)', ['admin', 1, 0])
>>> q.execute()
1
```

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

Return type a `RawQuery` for the given `Model`

example:

```
>>> q = User.raw('select id, username from users')
>>> for user in q:
...     print user.id, user.username
```

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

Parameters **fail_silently** – 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.

example:

```
>>> database.connect()
>>> SomeModel.create_table() # <-- creates the table for SomeModel
```

classmethod **`drop_table`**([*fail_silently=False*])

Parameters **fail_silently** – If set to `True`, the query will check for the existence of the table before attempting to remove.

Drop the table for the given model.

Note: Cascading deletes are not handled by this method, nor is the removal of any constraints.

classmethod **`table_exists`**()

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

1.6 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-level attributes. This should look familiar to users of the django framework. Here's an example:

```
from peewee import *

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

There is one special type of field, `ForeignKeyField`, which allows you to expose 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
```

1.6.1 Field types table

Parameters accepted by all field types and their default values:

- `null = False` – boolean indicating whether null values are allowed to be stored
- `db_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
- `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
- `choices = None` – an optional iterable containing 2-tuples of `value`, `display`

Field Type	Sqlite	Postgresql	MySQL
`CharField`	varchar	varchar	varchar
`TextField`	text	text	longtext
`DateTimeField`	datetime	timestamp	datetime
`IntegerField`	integer	integer	integer
`BooleanField`	smallint	boolean	bool
`FloatField`	real	real	real
`DoubleField`	real	double precision	double precision
`BigIntegerField`	integer	bigint	bigint
`DecimalField`	decimal	numeric	numeric
`PrimaryKeyField`	integer	serial	integer
`ForeignKeyField`	integer	integer	integer
`DateField`	date	date	date
`TimeField`	time	time	time

Some fields take special parameters...

Field type	Special Parameters
`CharField`	`max_length`
`DateTimeField`	`formats`
`DateField`	`formats`
`TimeField`	`formats`
`DecimalField`	`max_digits`, `decimal_places`, `auto_round`, `rounding`
`PrimaryKeyField`	`column_class`
`ForeignKeyField`	`to`, `related_name`, `cascade`, `extra`

A note on validation

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.

1.6.2 Self-referential Foreign Keys

Since the class is not available at the time the field is declared, when creating a self-referential foreign key pass in 'self' as the "to" relation:

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

1.6.3 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:

```
for student in Student.select().join(StudentCourse).join(Course).where(name='math'):
    print student.name
```

You could also express this as:

```
for student in Student.filter(studentcourse_set__course__name='math'):
    print student.name
```

To query what classes a given student is enrolled in:

```
for course in Course.select().join(StudentCourse).join(Student).where(name='da vinci'):
    print course.name

# or, similarly
for course in Course.filter(studentcourse_set__student__name='da vinci'):
    print course.name
```

1.6.4 Non-integer Primary Keys

First of all, let me say that I do not think using non-integer primary keys is a good idea. The cost in storage is higher, the index lookups will be slower, and foreign key joins will be more expensive. That being said, here is how you can use non-integer pks in peewee.

```
from peewee import Model, PrimaryKeyField, VarCharColumn

class UUIDModel(Model):
    # explicitly declare a primary key field, and specify the class to use
    id = PrimaryKeyField(column_class=VarCharColumn)
```

Auto-increment IDs are, as their name says, automatically generated for you when you insert a new row into the database. The way peewee determines whether to do an `INSERT` versus an `UPDATE` comes down to checking whether the primary key field is `None`. If `None`, it will do an insert, otherwise it does an update on the existing 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`:

```
inst = UUIDModel(id=str(uuid.uuid4()))
inst.save() # <-- WRONG!!  this will try to do an update

inst.save(force_insert=True) # <-- CORRECT

# to update the instance after it has been saved once
inst.save()
```

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

1.6.5 Field class API

class **`Field`**

The base class from which all other field types extend.

`__init__`(*null=False, db_index=False, unique=False, verbose_name=None, help_text=None, db_column=None, default=None, choices=None, *args, **kwargs*)

Parameters

- **null** – this column can accept `None` or `NULL` values
- **db_index** – create an index for this column when creating the table
- **unique** – create a unique index for this column when creating the table
- **verbose_name** – specify a "verbose name" for this field, useful for metadata purposes
- **help_text** – specify some instruction text for the usage/meaning of this field
- **db_column** – column class to use for underlying storage
- **default** – a value to use as an uninitialized default
- **choices** – an iterable of 2-tuples mapping `value` to `display`

`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

lookup_value(*lookup_type*, *value*)

Parameters

- **lookup_type** – a peewee lookup type, such as 'eq' or 'contains'
- **value** – a python data type

Return type data type converted for use when querying

class_prepared()

Simple hook for `Field` classes to indicate when the `Model` class the field exists on has been created.

class **CharField**

Stores: small strings (0-255 bytes)

class **TextField**

Stores: arbitrarily large strings

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

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

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

class **IntegerField**

Stores: integers

class **BooleanField**
Stores: `True` / `False`

class **FloatField**
Stores: floating-point numbers

class **DecimalField**
Stores: decimal numbers

class **PrimaryKeyField**
Stores: auto-incrementing integer fields suitable for use as primary key by default, though other types of data can be stored by specifying a column_class. See *notes on non-integer primary keys*.

__init__(*column_class*[, ...])

Parameters **column_class** – a reference to a subclass of `Column` to use for the underlying storage, defaults to `PrimaryKeyColumn`.

class **ForeignKeyField**
Stores: relationship to another model

__init__(*to*[, *related_name=None*[, ...]])

Parameters

- **to** – related `Model` class or the string 'self' if declaring a self-referential foreign key
- **related_name** – attribute to expose on related model

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

class Entry(Model):
    blog = ForeignKeyField(Blog, related_name='entries')
    title = CharField()
    content = TextField()

# "blog" attribute
>>> some_entry.blog
<Blog: My Awesome Blog>

# "entries" related name attribute
>>> for entry in my_awesome_blog.entries:
...     print entry.title
Some entry
Another entry
Yet another entry
```

1.7 Querying API

1.7.1 Constructing queries

Queries in peewee are constructed one piece at a time.

The "pieces" of a peewee query are generally representative of clauses you might find in a SQL query. Most methods are chainable, so you build your query up one clause at a time. This way, rather complex queries are possible.

Here is a barebones select query:

```
>>> user_q = User.select() # <-- query is not executed
>>> user_q
<peewee.SelectQuery object at 0x7f6b0810c610>

>>> [u.username for u in user_q] # <-- query is evaluated here
[u'admin', u'staff', u'editor']
```

We can build up the query by adding some clauses to it:

```
>>> user_q = user_q.where(username__in=['admin', 'editor'])
>>> user_q = user_q.order_by(('username', 'desc'))
>>> [u.username for u in user_q] # <-- query is re-evaluated here
[u'editor', u'admin']
```

Django-style queries

If you are already familiar with the Django ORM, you can construct `SelectQuery` instances using the familiar "double-underscore" syntax to generate the proper `JOIN`s and `WHERE` clauses.

Using python operators to query

You can use python operators to construct queries. This is possible by overloading operators on field instances.

Comparing the three methods of querying

Examples shown are "default", "django" and "python operators".

Get active users:

```
User.select().where(active=True)

User.filter(active=True)

User.select().where(User.active==True)
```

Get users who are either staff or superusers:

```
User.select().where(Q(is_staff=True) | Q(is_superuser=True))

User.filter(Q(is_staff=True) | Q(is_superuser=True))

User.select().where((User.is_staff==True) | (User.is_superuser==True))
```

Get tweets by user named "charlie":

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

Tweet.filter(user__username='charlie')

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

Get tweets by staff or superusers (assumes FK relationship):

```
Tweet.select().join(User).where(
    Q(is_staff=True) | Q(is_superuser=True)
)
```

```
Tweet.filter(Q(user__is_staff=True) | Q(user__is_superuser=True))

Tweet.select().join(User).where(
    (User.is_staff==True) | (User.is_superuser==True)
)
```

1.7.2 Where clause

All queries except `InsertQuery` support the `where()` method. If you are familiar with Django's ORM, it is analagous to the `filter()` method.

```
>>> User.select().where(is_staff=True).sql()
('SELECT * FROM user WHERE is_staff = ?', [1])
```

Note: `User.select()` is equivalent to `SelectQuery(User)`.

The `where()` method acts on the `Model` that is the current "query context". This is either:

- the model the query class was initialized with
- the model most recently JOINed on

Here is an example using JOINs:

```
>>> User.select().where(is_staff=True).join(Blog).where(status=LIVE)
```

This query grabs all staff users who have a blog that is "LIVE". This does the opposite, grabs all the blogs that are live whose author is a staffer:

```
>>> Blog.select().where(status=LIVE).join(User).where(is_staff=True)
```

Note: to `join()` from one model to another there must be a `ForeignKeyField` linking the two.

Another way to write the above query would be:

```
>>> Blog.select().where(
...     status=LIVE,
...     user__in=User.select().where(is_staff=True)
... )
```

The above bears a little bit of explanation. First off the SQL generated will not perform any explicit `JOIN` - it will rather use a subquery in the `WHERE` clause:

```
# using subqueries
SELECT * FROM blog
WHERE (
    status = ? AND
    user_id IN (
        SELECT t1.id FROM user AS t1 WHERE t1.is_staff = ?
    )
)
```

And here it is using joins:

```
# using joins
SELECT t1.* FROM blog AS t1
INNER JOIN user AS t2
    ON t1.user_id = t2.id
WHERE
    t1.status = ? AND
    t2.is_staff = ?
```

Column lookups

The other bit that's unique about the query is that it specifies `"user__in"`. Users familiar with Django will recognize this syntax - lookups other than "=" are signified by a double-underscore followed by the lookup type. The following lookup types are available in peewee:

`__eq`: x = y, the default

`__lt`: x < y

`__lte`: x <= y

`__gt`: x > y

`__gte`: x >= y

`__ne`: x != y

`__is`: x IS y, used for testing against NULL values

`__contains`: case-sensitive check for substring

`__icontains`: case-insensitive check for substring

`__startswith`: case-sensitive check for string prefix

`__istartswith`: case-insensitive check for string prefix

`__in`: x IN y, where y is either a list of values or a `SelectQuery`

Python operator overloads

If you are querying using python operator overloading, different comparisons are expressed using python operators. The following lookups are supported:

`==`: x = y

`<`: x < y

`<=`: x <= y

`>`: x > y

`>=`: x >= y

`!=`: x != y

`*`: case-sensitive check for substring

`**`: case-insensitive check for substring

`^`: case-insensitive check for string prefix

`>>`: x IS (NOT) NULL, depending on if y is True or False

`<<`: x IN y, where y is either a list of values or a `SelectQuery`

1.7.3 Performing advanced queries

As you may have noticed, all the examples up to now have shown queries that combine multiple clauses with "AND". Taking another page from Django's ORM, peewee allows the creation of arbitrarily complex queries using a special notation called `Q` objects.

```
>>> sq = User.select().where(Q(is_staff=True) | Q(is_superuser=True))
>>> print sq.sql()[0]
SELECT * FROM user WHERE (is_staff = ? OR is_superuser = ?)
```

`Q` objects can be combined using the bitwise "or" and "and" operators. In order to negate a `Q` object, use the bitwise "invert" operator:

```
>>> staff_users = User.select().where(is_staff=True)
>>> Blog.select().where(~Q(user__in=staff_users))
```

This query generates the following SQL:

```
SELECT * FROM blog
WHERE
    NOT user_id IN (
        SELECT t1.id FROM user AS t1 WHERE t1.is_staff = ?
    )
```

Rather complex lookups are possible:

```
>>> sq = User.select().where(
...     (Q(is_staff=True) | Q(is_superuser=True)) &
...     (Q(join_date__gte=datetime(2009, 1, 1)) | Q(join_date__lt=datetime(2005, 1 1)))
... )
>>> print sq.sql()[0] # cleaned up
SELECT * FROM user
WHERE (
    (is_staff = ? OR is_superuser = ?) AND
    (join_date >= ? OR join_date < ?)
)
```

This query selects all staff or super users who joined after 2009 or before 2005.

Note: If you need more power, check out `RawQuery`

Comparing against column data

Suppose you have a model that looks like the following:

```
class WorkerProfiles(Model):
    salary = IntegerField()
    desired = IntegerField()
```

What if we want to query `WorkerProfiles` to find all the rows where "salary" is greater than "desired" (maybe you want to find out who may be looking for a raise)?

To solve this problem, peewee borrows the notion of `F` objects from the django orm. An `F` object allows you to query against arbitrary data present in another column:

```
WorkerProfile.select().where(salary__gt=F('desired'))
```

That's it. If the other column exists on a model that is accessed via a JOIN, you will need to specify that model as the second argument to the `F` object. Let's supposed that the "desired" salary exists on a separate model:

```
WorkerProfile.select().join(Desired).where(desired_salary__lt=F('salary', WorkerProfile))
```

Atomic updates

The `F` object also works for updating data. Suppose you cache counts of tweets for every user in a special table to avoid an expensive COUNT() query. You want to update the cache table every time a user tweets, but do so atomically:

```
cache_row = CacheCount.get(user=some_user)
update_query = cache_row.update(tweet_count=F('tweet_count') + 1)
update_query.execute()
```

Aggregating records

Suppose you have some blogs and want to get a list of them along with the count of entries in each. First I will show you the shortcut:

```
query = Blog.select().annotate(Entry)
```

This is equivalent to the following:

```
query = Blog.select({
    Blog: ['*'],
    Entry: [Count('id')],
}).group_by(Blog).join(Entry)
```

The resulting query will return `Blog` objects with all their normal attributes plus an additional attribute 'count' which will contain the number of entries. By default it uses an inner join if the foreign key is not nullable, which means blogs without entries won't appear in the list. To remedy this, manually specify the type of join to include blogs with 0 entries:

```
query = Blog.select().join(Entry, 'left outer').annotate(Entry)
```

You can also specify a custom aggregator:

```
query = Blog.select().annotate(Entry, peewee.Max('pub_date', 'max_pub_date'))
```

Conversely, sometimes you want to perform an aggregate query that returns a scalar value, like the "max id". Queries like this can be executed by using the `aggregate()` method:

```
max_id = Blog.select().aggregate(Max('id'))
```

SQL Functions, "Raw expressions" and the R() object

If you've been reading in order, you will have already seen the `Q` and `F` objects. The `R` object is the final query helper and its purpose is to allow you to express arbitrary expressions as part of your structured query without having to result to using a `RawQuery`.

Selecting users whose username begins with "a":

```
# select the users' id, username and the first letter of their username, lower-cased
query = User.select(['id', 'username', R('LOWER(SUBSTR(username, 1, 1))', 'first_letter')])

# now filter this list to include only users whose username begins with "a"
```

```
a_users = query.where(R('first_letter=%s', 'a'))

>>> for user in a_users:
...    print user.first_letter, user.username

a alpha
A Alton
```

This same functionality could be easily exposed as part of the where clause, the only difference being that the first letter is not selected and therefore not an attribute of the model instance:

```
a_users = User.filter(R('LOWER(SUBSTR(username, 1, 1)) = %s', 'a'))
```

We can query for multiple values using `R` objects, for example selecting users whose usernames begin with a range of letters "b" through "d":

```
letters =  ('b', 'c', 'd')
bcd_users = User.filter(R('LOWER(SUBSTR(username, 1, 1)) IN (%s, %s, %s)', *letters))
```

We can write subqueries as part of a `SelectQuery`, for example counting the number of entries on a blog:

```
entry_query = R('(SELECT COUNT(*) FROM entry WHERE entry.blog_id=blog.id)', 'entry_count')
blogs = Blog.select(['id', 'title', entry_query]).order_by(('entry_count', 'desc'))

for blog in blogs:
    print blog.title, blog.entry_count
```

It is also possible to use subqueries as part of a where clause, for example finding blogs that have no entries:

```
no_entry_query = R('NOT EXISTS (SELECT * FROM entry WHERE entry.blog_id=blog.id)')
blogs = Blog.filter(no_entry_query)

for blog in blogs:
    print blog.title, ' has no entries'
```

Saving Queries by Selecting Related Models

Returning to my favorite models, `Blog` and `Entry`, between which there is a `ForeignKeyField`, a common pattern might be to display a list of the latest 10 entries with some info about the blog they're on as well. We can do this pretty easily:

```
for entry in Entry.select().order_by(('pub_date', 'desc')).limit(10):
    print '%s, posted on %s' % (entry.title, entry.blog.title)
```

Looking at the query log, though, this will cause 11 queries:

- 1 query for the entries
- 1 query for every related blog (10 total)

This can be optimized into one query very easily, though:

```
entries = Entry.select({
    Entry: ['*'],
    Blog: ['*'],
}).order_by(('pub_date', 'desc')).join(Blog)

for entry in entries.limit(10):
    print '%s, posted on %s' % (entry.title, entry.blog.title)
```

Will cause only one query that looks something like this:

```
SELECT t1.pk, t1.title, t1.content, t1.pub_date, t1.blog_id, t2.id, t2.title
FROM entry AS t1
INNER JOIN blog AS t2
    ON t1.blog_id = t2.id
ORDER BY t1.pub_date desc
LIMIT 10
```

peewee will handle constructing the objects and you can access them as you would normally.

Note: Note in the above example the call to `.join(Blog)`

This works for following objects "up" the chain, i.e. following foreign key relationships. The reverse is not true, however – you cannot issue a single query and get all related sub-objects, i.e. list blogs and prefetch all related entries. This *can* be done by fetching all entries (with related blog data), then reconstructing the blogs in python, but is not provided as part of peewee. For a detailed discussion of working around this, see the discussion here.

Speeding up simple select queries

Simple select queries can get a performance boost (especially when iterating over large result sets) by calling `naive()`. This method simply patches all attributes directly from the cursor onto the model. For simple queries this should have no noticeable impact. The main difference is when multiple tables are queried, as in the previous example:

```
# above example
entries = Entry.select({
    Entry: ['*'],
    Blog: ['*'],
}).order_by(('pub_date', 'desc')).join(Blog)

for entry in entries.limit(10):
    print '%s, posted on %s' % (entry.title, entry.blog.title)
```

And here is how you would do the same if using a naive query:

```
# very similar query to the above -- main difference is we're
# aliasing the blog title to "blog_title"
entries = Entry.select({
    Entry: ['*'],
    Blog: [('title', 'blog_title')],
}).order_by(('pub_date', 'desc')).join(Blog)

entries = entries.naive()

# now instead of calling "entry.blog.title" the blog's title
# is exposed directly on the entry model as "blog_title" and
# no blog instance is created
for entry in entries.limit(10):
    print '%s, posted on %s' % (entry.title, entry.blog_title)
```

1.7.4 Query evaluation

In order to execute a query, it is *always* necessary to call the `execute()` method.

To get a better idea of how querying works let's look at some example queries and their return values:

```
>>> dq = User.delete().where(active=False) # <-- returns a DeleteQuery
>>> dq
<peewee.DeleteQuery object at 0x7fc866ada4d0>
>>> dq.execute() # <-- executes the query and returns number of rows deleted
3

>>> uq = User.update(active=True).where(id__gt=3) # <-- returns an UpdateQuery
>>> uq
<peewee.UpdateQuery object at 0x7fc865beff50>
>>> uq.execute() # <-- executes the query and returns number of rows updated
2

>>> iq = User.insert(username='new user') # <-- returns an InsertQuery
>>> iq
<peewee.InsertQuery object at 0x7fc865beff10>
>>> iq.execute() # <-- executes query and returns the new row's PK
3

>>> sq = User.select().where(active=True) # <-- returns a SelectQuery
>>> sq
<peewee.SelectQuery object at 0x7fc865b7a510>
>>> qr = sq.execute() # <-- executes query and returns a QueryResultWrapper
>>> qr
<peewee.QueryResultWrapper object at 0x7fc865b7a6d0>
>>> [u.id for u in qr]
[1, 2, 3, 4, 7, 8]
>>> [u.id for u in qr] # <-- re-iterating over qr does not re-execute query
[1, 2, 3, 4, 7, 8]

>>> [u.id for u in sq] # <-- as a shortcut, you can iterate directly over
>>>                    #     a SelectQuery (which uses a QueryResultWrapper
>>>                    #     behind-the-scenes)
[1, 2, 3, 4, 7, 8]
```

Note: Iterating over a `SelectQuery` will cause it to be evaluated, but iterating over it multiple times will not result in the query being executed again.

1.7.5 QueryResultWrapper

As I hope the previous bit showed, `Delete`, `Insert` and `Update` queries are all pretty straightforward. `Select` queries are a little bit tricky in that they return a special object called a `QueryResultWrapper`. The sole purpose of this class is to allow the results of a query to be iterated over efficiently. In general it should not need to be dealt with explicitly.

The preferred method of iterating over a result set is to iterate directly over the `SelectQuery`, allowing it to manage the `QueryResultWrapper` internally.

1.7.6 SelectQuery

class `SelectQuery`

By far the most complex of the 4 query classes available in peewee. It supports `JOIN` operations on other tables, aggregation via `GROUP BY` and `HAVING` clauses, ordering via `ORDER BY`, and can be iterated and sliced to return only a subset of results.

__init__(*model*, *query=None*)

Parameters

- **model** – a `Model` class to perform query on
- **query** – either a dictionary, keyed by model with a list of columns, or a string of columns

If no query is provided, it will default to `'*'`. this parameter can be either a dictionary or a string:

```
>>> sq = SelectQuery(Blog, {Blog: ['id', 'title']})
>>> sq = SelectQuery(Blog, {
...     Blog: ['*'],
...     Entry: [peewee.Count('id')]
... }).group_by('id').join(Entry)
>>> print sq.sql()[0] # formatted
SELECT t1.*, COUNT(t2.id) AS count
FROM blog AS t1
INNER JOIN entry AS t2
    ON t1.id = t2.blog_id
GROUP BY t1.id

>>> sq = SelectQuery(Blog, 'id, title')
>>> print sq.sql()[0]
SELECT id, title FROM blog
```

filter(**args*, ***kwargs*)

Parameters

- **args** – a list of `Q` or `Node` objects
- **kwargs** – a mapping of column + lookup to value, e.g. "age__gt=55"

Return type a `SelectQuery` instance

Provides a django-like syntax for building a query. The key difference between `filter()` and `where()` is that `filter` supports traversing joins using django's "double-underscore" syntax:

```
>>> sq = SelectQuery(Entry).filter(blog__title='Some Blog')
```

This method is chainable:

```
>>> base_q = User.filter(active=True)
>>> some_user = base_q.filter(username='charlie')
```

get(**args*, ***kwargs*)

Parameters

- **args** – a list of `Q` or `Node` objects
- **kwargs** – a mapping of column + lookup to value, e.g. "age__gt=55"

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(active=True)
>>> try:
...     user = active.get(username=username, password=password)
... except User.DoesNotExist:
...     user = None
```

This method is also exposed via the `Model` api:

```
>>> user = User.get(username=username, password=password)
```

`where`(**args*, ***kwargs*)

Parameters

- **args** – a list of `Q` or `Node` objects
- **kwargs** – a mapping of column + lookup to value, e.g. "age__gt=55"

Return type a `SelectQuery` instance

Calling `where()` will act on the model that is currently the `query context`. Unlike `filter()`, only columns from the current query context are exposed:

```
>>> sq = SelectQuery(Blog).where(title='some title', author=some_user)
>>> sq = SelectQuery(Blog).where(Q(title='some title') | Q(title='other title'))
```

Note: `where()` calls are chainable

`join`(*model*, *join_type=None*, *on=None*, *alias=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
- **on** – if multiple foreign keys exist between two models, this parameter is a string containing the name of the ForeignKeyField to join on.
- **alias** – if provided, will be the name used to alias columns from this table in query

Return type a `SelectQuery` instance

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

```
>>> sq = SelectQuery(Blog).join(Entry).where(title='Some Entry')
>>> sq = SelectQuery(User).join(Relationship, on='to_user_id').where(from_user=self)
```

`naive`()

Return type `SelectQuery`

indicates that this query 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.

Note: this can provide a significant speed improvement when doing simple iteration over a large result set.

`switch`(*model*)

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

Return type a `SelectQuery` instance

Switches the `query context` to the given model. Raises an exception if the model has not been selected or joined on previously.

```
>>> sq = SelectQuery(Blog).join(Entry).switch(Blog).where(title='Some Blog')
```

count()

Return type an integer representing the number of rows in the current query

```
>>> sq = SelectQuery(Blog)
>>> sq.count()
45 # <-- number of blogs
>>> sq.where(status=DELETED)
>>> sq.count()
3 # <-- number of blogs that are marked as deleted
```

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(active=True)
>>> if sq.where(username=username, password=password).exists():
...     authenticated = True
```

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. `Max('pub_date', 'max_pub')`

Return type `SelectQuery`

Annotate a query with an aggregation performed on a related model, for example, "get a list of blogs with the number of entries on each":

```
>>> Blog.select().annotate(Entry)
```

if `aggregation` is None, it will default to `Count(related_model, 'count')`, but can be anything:

```
>>> blog_with_latest = Blog.select().annotate(Entry, Max('pub_date', 'max_pub'))
```

Note: If the `ForeignKeyField` is `nullable`, then a `LEFT OUTER` join will be used, otherwise the join is an `INNER` join. If an `INNER` join is used, in the above example blogs with no entries would not be returned. To avoid this, you can explicitly join before calling `annotate()`:

```
>>> Blog.select().join(Entry, 'left outer').annotate(Entry)
```

aggregate(*aggregation*)

Parameters aggregation – a function specifying what aggregation to perform, for example `Max('id')`. This can be a 3-tuple if you would like to perform a custom aggregation: `("Max", "id", "max_id")`.

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.

group_by(*clause*)

Parameters **clause** – either a single field name or a list of field names, in which case it takes its context from the current query_context. it can *also* be a model class, in which case all that models fields will be included in the `GROUP BY` clause

Return type `SelectQuery`

```
>>> # get a list of blogs with the count of entries each has
>>> sq = Blog.select({
...     Blog: ['*'],
...     Entry: [Count('id')]
... }).group_by('id').join(Entry)

>>> # slightly more complex, get a list of blogs ordered by most recent pub_date
>>> sq = Blog.select({
...     Blog: ['*'],
...     Entry: [Max('pub_date', 'max_pub_date')],
... }).join(Entry)
>>> # now, group by the entry's blog id, followed by all the blog fields
>>> sq = sq.group_by('blog_id').group_by(Blog)
>>> # finally, order our results by max pub date
>>> sq = sq.order_by(peewee.desc('max_pub_date'))
```

having(*clause*)

Parameters **clause** – Expression to use as the `HAVING` clause

Return type `SelectQuery`

```
>>> sq = Blog.select({
...     Blog: ['*'],
...     Entry: [Count('id', 'num_entries')]
... }).group_by('id').join(Entry).having('num_entries > 10')
```

order_by(**clauses*)

Parameters **clauses** – Expression(s) to use as the `ORDER BY` clause, see notes below

Return type `SelectQuery`

Note: Adds the provided clause (a field name or alias) to the query's `ORDER BY` clause. It can be either a single field name, in which case it will apply to the current query context, or a 2- or 3-tuple.

The 2-tuple can be either `(Model, 'field_name')` or `('field_name', 'ASC'/'DESC')`.

The 3-tuple is `(Model, 'field_name', 'ASC'/'DESC')`.

If the field is not found on the model evaluated against, it will be treated as an alias.

example:

```
>>> sq = Blog.select().order_by('title')
>>> sq = Blog.select({
...     Blog: ['*'],
...     Entry: [Max('pub_date', 'max_pub')]
... }).join(Entry).order_by(desc('max_pub'))
```

slightly more complex example:

```
>>> sq = Entry.select().join(Blog).order_by(
...     (Blog, 'title'), # order by blog title ascending
...     (Entry, 'pub_date', 'DESC'), # then order by entry pub date desc
... )
```

check out how the `query context` applies to ordering:

```
>>> blog_title = Blog.select().order_by('title').join(Entry)
>>> print blog_title.sql()[0]
SELECT t1.* FROM blog AS t1
INNER JOIN entry AS t2
    ON t1.id = t2.blog_id
ORDER BY t1.title

>>> entry_title = Blog.select().join(Entry).order_by('title')
>>> print entry_title.sql()[0]
SELECT t1.* FROM blog AS t1
INNER JOIN entry AS t2
    ON t1.id = t2.blog_id
ORDER BY t2.title # <-- note that it's using the title on Entry this time
```

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`

applies a `LIMIT` and `OFFSET` to the query.

```
>>> Blog.select().order_by('username').paginate(3, 20) # <-- get blogs 41-60
```

distinct()

Return type `SelectQuery`

indicates that this query should only return distinct rows. results in a `SELECT DISTINCT` query.

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:

```
>>> for user in User.select().where(active=True):
...     print user.username
```

1.7.7 UpdateQuery

class **UpdateQuery**

Used for updating rows in the database.

__init__(*model*, ***kwargs*)

Parameters

- **model** – `Model` class on which to perform update
- **kwargs** – mapping of field/value pairs containing columns and values to update

```
>>> uq = UpdateQuery(User, active=False).where(registration_expired=True)
>>> print uq.sql()
('UPDATE user SET active=? WHERE registration_expired = ?', [0, True])

>>> atomic_update = UpdateQuery(User, message_count=F('message_count') + 1).where(id=3)
>>> print atomic_update.sql()
('UPDATE user SET message_count=(message_count + 1) WHERE id = ?', [3])
```

`where`(**args*, ***kwargs*)

Parameters

- **args** – a list of `Q` or `Node` objects
- **kwargs** – a mapping of column + lookup to value, e.g. "age__gt=55"

Return type a `UpdateQuery` instance

Note: `where()` calls are chainable

`execute`()

Return type Number of rows updated

Performs the query

1.7.8 DeleteQuery

class **`DeleteQuery`**

Deletes rows of the given model.

Note: It will *not* traverse foreign keys or ensure that constraints are obeyed, so use it with care.

`__init__`(*model*)

creates a `DeleteQuery` instance for the given model:

```
>>> dq = DeleteQuery(User).where(active=False)
>>> print dq.sql()
('DELETE FROM user WHERE active = ?', [0])
```

`where`(**args*, ***kwargs*)

Parameters

- **args** – a list of `Q` or `Node` objects
- **kwargs** – a mapping of column + lookup to value, e.g. "age__gt=55"

Return type a `DeleteQuery` instance

Note: `where()` calls are chainable

`execute`()

Return type Number of rows deleted

Performs the query

1.7.9 InsertQuery

class **InsertQuery**

Creates a new row for the given model.

__init__(*model*, ***kwargs*)

creates an `InsertQuery` instance for the given model where kwargs is a dictionary of field name to value:

```
>>> iq = InsertQuery(User, username='admin', password='test', active=True)
>>> print iq.sql()
('INSERT INTO user (username, password, active) VALUES (?, ?, ?)', ['admin', 'test', 1])
```

execute()

Return type primary key of the new row

Performs the query

1.7.10 RawQuery

class **RawQuery**

Allows execution of an arbitrary `SELECT` query and returns instances of the model via a `QueryResultsWrapper`.

__init__(*model*, *query*, **params*)

creates a `RawQuery` instance for the given model which, when executed, will run the given query with the given parameters and return model instances:

```
>>> rq = RawQuery(User, 'SELECT * FROM users WHERE username = ?', 'admin')
>>> for obj in rq.execute():
...     print obj
<User: admin>
```

execute()

Return type a `QueryResultWrapper` for iterating over the result set. The results are instances of the given model.

Performs the query

1.8 Databases

Below the `Model` level, peewee uses an abstraction for representing the database. The `Database` is responsible for establishing and closing connections, making queries, and gathering information from the database.

The `Database` in turn uses another abstraction called an `BaseAdapter`, which is backend-specific and encapsulates functionality specific to a given db driver. Since there is some difference in column types across database engines, this information also resides in the adapter. The adapter is responsible for smoothing out the quirks of each database driver to provide a consistent interface, for example sqlite uses the question-mark ”?” character for parameter interpolation, while all the other backends use “%s”.

For a high-level overview of working with transactions, check out the *transactions cookbook*.

For notes on deferring instantiation of database, for example if loading configuration at run-time, see the notes on *deferring initialization*.

Note: The internals of the `Database` and `BaseAdapter` will be of interest to anyone interested in adding support for another database driver.

1.8.1 Writing a database driver

Peewee currently supports Sqlite, MySQL and Postgresql. 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*
- *Cursor.fetchmany*

These methods are generally wrapped up in higher-level abstractions and exposed by the `Database` and `BaseAdapter`, 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 which I'm tinkering with adding support for.

Note: In later versions of peewee, the db-api 2.0 methods may be further abstracted out to add support for drivers that don't conform to the spec.

Getting down to it, writing some classes

There are two classes you will want to implement at the very least:

- **`BaseAdapter` - handles low-level functionality like opening and closing** connections to the database, as well as describing the features provided by the database engine
- **`Database` - higher-level interface that executes queries, manage** transactions, and can introspect the underlying db.

Let's say we want to add support for a fictitious "FooDB" which has an open-source python driver that uses the DB-API 2.0.

The Adapter

The adapter provides a bridge between the driver and peewee's higher-level database class which is responsible for executing queries.

```
from peewee import BaseAdapter
import foodb # our fictional driver


class FooAdapter(BaseAdapter):
    def connect(self, database, **kwargs):
        return foodb.connect(database, **kwargs)
```

Now we want to create a mapping that exposes the operations our database engine supports. These are the operations that a user perform when building out the `WHERE` clause of a given query.

```
class FooAdapter(BaseAdapter):
    operations = {
        'lt': '< %s',
        'lte': '<= %s',
        'gt': '> %s',
        'gte': '>= %s',
        'eq': '= %s',
        'ne': '!= %s',
        'in': 'IN (%s)',
        'is': 'IS %s',
        'isnull': 'IS NULL',
        'between': 'BETWEEN %s AND %s',
        'icontains': 'ILIKE %s',
        'contains': 'LIKE %s',
        'istartswith': 'ILIKE %s',
        'startswith': 'LIKE %s',
    }

    def connect(self, database, **kwargs):
        return foodb.connect(database, **kwargs)
```

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

- last insert id and number of rows modified
- specifying characters used for string interpolation and quoting identifiers, for instance, sqlite uses "?" for interpolation and MySQL uses a backtick for quoting
- modifying user input for various lookup types, for instance a "LIKE" query will surround the incoming phrase with "%" characters.

The database class

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. Each database must specify a `BaseAdapter` subclass, so our database will need to specify the `FooAdapter` we just defined:

```
from peewee import Database

class FooDatabase(Database):
    def __init__(self, database, **connect_kwargs):
        super(FooDatabase, self).__init__(FooAdapter(), database, **connect_kwargs)
```

This 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 __init__(self, database, **connect_kwargs):
        super(FooDatabase, self).__init__(FooAdapter(), database, **connect_kwargs)

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

    def get_indexes_for_table(self, table):
        res = self.execute('SHOW INDEXES IN %s;' % self.quote_name(table))
        rows = sorted([(r[2], r[1] == 0) for r in res.fetchall()])
        return rows
```

There is a good deal of functionality provided by the Database class that is not covered here. Refer to the documentation below 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.

Using our new database

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

```
from peewee import *
from foodb_ext import FooDatabase

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

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

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

1.8.2 Database and its subclasses

class `Database`

A high-level api for working with the supported database engines. `Database` provides a wrapper around some of the functions performed by the `Adapter`, in addition providing support for:

- execution of SQL queries
- creating and dropping tables and indexes

`__init__`(*adapter*, *database*[, *threadlocals=False*[, *autocommit=True*[, ***connect_kwargs*]]])

Parameters

- **adapter** – an instance of a `BaseAdapter` subclass
- **database** – the name of the database (or filename if using sqlite)
- **threadlocals** – whether to store connections in a threadlocal
- **autocommit** – automatically commit every query executed by calling `execute()`

- **connect_kwargs** – any arbitrary parameters to pass to the database driver when connecting

---

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

---

`init`(*database*[, ***connect_kwargs*])
If the database was instantiated with database=None, the database is said to be in a 'deferred' state (see *notes*) – if this is the case, you can initialize it at any time by calling the `init` method.

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: If you initialized with `threadlocals=True`, then this will store the connection inside a thread-local, ensuring that connections are not shared across threads.

`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.

`get_conn`()

Return type a connection to the database, creates one if does not exist

`get_cursor`()

Return type a cursor for executing queries

`set_autocommit`(*autocommit*)

Parameters autocommit – a boolean value indicating whether to turn on/off autocommit **for the current connection**

`get_autocommit`()

Return type a boolean value indicating whether autocommit is on **for the current connection**

`execute`(*sql*[, *params=None*])

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.

`commit`()
Call `commit()` on the active connection, committing the current transaction

rollback()

Call `rollback()` on the active connection, rolling back the current transaction

commit_on_success(*func*)

Decorator that wraps the given function in a single transaction, which, upon success will be committed. If an error is raised inside the function, the transaction will be rolled back and the error will be re-raised.

Parameters func – function to decorate

```
@database.commit_on_success
def transfer_money(from_acct, to_acct, amt):
    from_acct.charge(amt)
    to_acct.pay(amt)
    return amt
```

transaction()

Return a context manager that executes statements in a transaction. If an error is raised inside the context manager, the transaction will be rolled back, otherwise statements are committed when exiting.

```
# delete a blog instance and all its associated entries, but
# do so within a transaction
with database.transaction():
    blog.delete_instance(recursive=True)
```

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

create_table(*model_class*[, *safe=False*])

Parameters

- **model_class** – `Model` class to create table for
- **safe** – if `True`, query will add a `IF NOT EXISTS` clause

create_index(*model_class*, *field_names*[, *unique=False*])

Parameters

- **model_class** – `Model` table on which to create index
- **field_name** – name of field(s) to create index on (a string or list)
- **unique** – whether the index should enforce uniqueness

create_foreign_key(*model_class*, *field*)

Parameters

- **model_class** – `Model` table on which to create foreign key index / constraint
- **field** – `Field` object

drop_table(*model_class*[, *fail_silently=False*])

Parameters

- **model_class** – `Model` table to drop
- **fail_silently** – if `True`, query will add a `IF EXISTS` clause

Note: Cascading drop tables are not supported at this time, so if a constraint exists that prevents a table being dropped, you will need to handle that in application logic.

`add_column_sql`(*model_class*, *field_name*)

Parameters

- **model_class** – `Model` which we are adding a column to
- **field_name** (*string*) – the name of the field we are adding

Return type SQL suitable for adding the column

Note: Adding a non-null column to a table with rows may cause an IntegrityError.

`rename_column_sql`(*model_class*, *field_name*, *new_name*)

Parameters

- **model_class** – `Model` instance
- **field_name** (*string*) – the current name of the field
- **new_name** (*string*) – new name for the field

Return type SQL suitable for renaming the column

Note: There must be a field instance named `field_name` at the time this SQL is generated.

Note: SQLite does not support renaming columns

`drop_column_sql`(*model_class*, *field_name*)

Parameters

- **model_class** – `Model` instance
- **field_name** (*string*) – the name of the field to drop

Note: SQLite does not support dropping columns

`create_sequence`(*sequence_name*)

Parameters **sequence_name** – name of sequence to create

Note: only works with database engines that support sequences

`drop_sequence`(*sequence_name*)

Parameters **sequence_name** – name of sequence to drop

Note: only works with database engines that support sequences

get_indexes_for_table(*table*)

Parameters **table** – the name of table to introspect

Return type a list of `(index_name, is_unique)` tuples

> **Warning:** Not implemented – implementations exist in subclasses

get_tables()

Return type a list of table names in the database

> **Warning:** Not implemented – implementations exist in subclasses

sequence_exists(*sequence_name*)

Rtype boolean

class **SqliteDatabase**(*Database*)
`Database` subclass that communicates to the "sqlite3" driver

class **MySQLDatabase**(*Database*)
`Database` subclass that communicates to the "MySQLdb" driver

class **PostgresqlDatabase**(*Database*)
`Database` subclass that communicates to the "psycopg2" driver

1.8.3 BaseAdapter and its subclasses

class **BaseAdapter**
The various subclasses of *BaseAdapter* provide a bridge between the high- level `Database` abstraction and the underlying python libraries like psycopg2. It also provides a way to unify the pythonic field types with the underlying column types used by the database engine.

The *BaseAdapter* provides two types of mappings: - mapping between filter operations and their database equivalents - mapping between basic field types and their database column types

The *BaseAdapter* also is the mechanism used by the `Database` class to: - handle connections with the database - extract information from the database cursor

operations = {'eq': '= %s'}
A mapping of query operation to SQL

interpolation = '%s'
The string used by the driver to interpolate query parameters

sequence_support = False
Whether the given backend supports sequences

reserved_tables = []
Table names that are reserved by the backend – if encountered in the application a warning will be issued.

get_field_types()

Return type a dictionary mapping "user-friendly field type" to specific column type, e.g. `{'string': 'VARCHAR', 'float': 'REAL', ... }`

get_field_type_overrides()

Return type a dictionary similar to that returned by `get_field_types()`.

Provides a mechanism to override any number of field types without having to override all of them.

connect (*database*, ***kwargs*)

Parameters

- **database** – string representing database name (or filename if using sqlite)
- **kwargs** – any keyword arguments to pass along to the database driver when connecting

Return type a database connection

close (*conn*)

Parameters **conn** – a database connection

Close the given database connection

lookup_cast (*lookup*, *value*)

Parameters

- **lookup** – a string representing the lookup type
- **value** – a python value that will be passed in to the lookup

Return type a converted value appropriate for the given lookup

Used as a hook when a specific lookup requires altering the given value, like for example when performing a LIKE query you may need to insert wildcards.

last_insert_id (*cursor*, *model*)

Return type most recently inserted primary key

rows_affected (*cursor*)

Return type number of rows affected by most recent query

class SqliteAdapter (*BaseAdapter*)
Subclass of `BaseAdapter` that works with the "sqlite3" driver

class MySQLAdapter (*BaseAdapter*)
Subclass of `BaseAdapter` that works with the "MySQLdb" driver

class PostgresqlAdapter (*BaseAdapter*)
Subclass of `BaseAdapter` that works with the "psycopg2" driver

1.9 Playhouse, a collection of addons

Peewee comes with numerous extras which I didn't really feel like including in the main source module, but which might be interesting to implementers or fun to mess around with.

1.9.1 apsw, an advanced sqlite driver

The `apsw_ext` module contains a database class suitable for use with the apsw sqlite driver. With apsw, it is possible to use some of the more advanced features of sqlite. It also offers better performance than pysqlite and finer-grained control over query execution. For more information on the differences between apsw and pysqlite, check the apsw docs.

Example usage

```
from apsw_ext import *

db = APSWDatabase(':memory:')

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

class SomeModel(BaseModel):
    col1 = CharField()
    col2 = DateTimeField()
    # etc, etc
```

apsw_ext API notes

class **APSWDatabase**(*database*, ***connect_kwargs*)

Parameters

- **database** (*string*) – filename of sqlite database
- **connect_kwargs** – keyword arguments passed to apsw when opening a connection

transaction([*lock_type='deferred'*])

Functions just like the `Database.transaction()` context manager, but accepts an additional parameter specifying the type of lock to use.

Parameters **lock_type** (*string*) – type of lock to use when opening a new transaction

class **APSWAdapter**(*timeout*)

Parameters **timeout** (*int*) – sqlite busy timeout in seconds (docs)

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

class **VirtualModel**

A model subclass suitable for creating virtual tables.

Note: You must specify the name for the extension module you wish to use

_extension_module

The name of the extension module to use with this virtual table

1.9.2 Postgresql Extensions (hstore, ltree)

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

- *hstore support*
- *ltree support*

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. hstore is great for storing JSON.

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(id=h.id)
>>> h_from_db.features
{'bath': '2 bath', 'garage': '2 cars'}
```

You can filter by keys or partial dictionary:

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

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

```
>>> query = House.update(features=hupdate('features', {'bath': '2.5 bath', 'sqft': '1100'}))
>>> query.where(id=h.id).execute()
1
>>> h = House.get(id=h.id)
>>> h.features
{'bath': '2.5 bath', 'garage': '2 cars', 'sqft': '1100'}
```

Or, alternatively an atomic delete:

```
>>> query = House.update(features=hdelete('features', 'bath'))
>>> query.where(id=h.id).execute()
1
>>> h = House.get(id=h.id)
>>> h.features
{'garage': '2 cars', 'sqft': '1100'}
```

Multiple keys can be deleted at the same time:

```
>>> query = House.update(features=hdelete('features', ['garage', 'sqft']))
```

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

```
>>> for h in House.select(['address', hkeys('features', 'keys')]):
...     print h.address, h.keys

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

>>> for h in House.select(['address', hvalues('features', 'vals')]):
...     print h.address, h.vals

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

>>> for h in House.select(['address', hmatrix('features', '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:

```
>>> for h in House.select(['address', hslice('features', 'garage_data', ['garage'])]):
...     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(['address', hexist('features', 'has_garage', 'garage')]):
...     print h.address, h.has_garage

123 Main St True

>>> for h in House.select().where(hexist('features', ['garage'])):
...     print h.address, h.features['garage'] # <-- just houses w/garage data
```

```
123 Main St 2 cars
```

ltree support

Postgresql ltree is a hierarchical data store. With ltree, you can store hierarchical structures as a series of labels separated by a delimiter (”.”).

Currently the `postgres_ext` module supports the following operations:

- store and retrieve label-trees
- very complex filtering by labels
- querying by prefix

using ltree

As with *hstore*, you will need to import the custom database class and the ltree functions from `playhouse.postgres_ext` (see above code snippet). Then, it is as simple as adding a `LTreeField` to your model:

```
class Category(BaseExtModel):
    name = CharField()
    path = LTreeField()
```

You can now store hierarchy information on Category instances. Let’s use the following data-set:

- **languages**
  - **dynamic**
    - python
    - ruby
  - **static**
    - c
    - c++
    - java

```
>>> Category.create(name='java', path='languages.static.java') # last one...
>>> Category.get(name='python').path
'languages.dynamic.python'
```

You can filter by path. Complex queries are possible, so please refer to the ltree documentation for details.

```
>>> show = lambda q: [l.name for l in q]
>>> show(Category.select().where(path__startswith='languages.dynamic'))
[u'dynamic', u'python', u'ruby']

>>> show(Category.select().where(path__lmatch='*.static.c*'))
[u'c', u'c++']

>>> show(Category.select().where(path__lmatch_text='(static | dynamic) & (p* | c*) & !cpp'))
[u'python', u'c']
```

You can select subtrees, label indices, and more:

```
>>> q = Category.select(['name', lsubpath('path', -1, 1, 'leaf')])
>>> for cat in q:
...     cat.leaf

languages
dynamic
static
python
ruby
c
cpp
java

>>> q = Category.select([lindex('path', 'static', 'static_pos')])
>>> [x.static_pos for x in q]
[-1, -1, 1, -1, -1, 1, 1, 1]

>>> q = Category.select([nlevel('path', 'depth')]).where(name='python')
>>> print q.get().depth
3
```

1.9.3 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:

```
pwiz.py -e postgresql -u postgres my_postgres_db > my_models.py
```

This will print a bunch of models to standard output. So you can do this:

```
pwiz.py -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

1.9.4 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, connect, post_save


class MyModel(Model):
    data = IntegerField()

@connect(post_save, sender=MyModel)
def on_save_handler(model_class, instance, created):
    put_data_in_cache(instance.data)
```

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:

```
@connect(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

connect (*signal*[, *sender=None*[, *name=None*]])
Function decorator that is an alias for a signal's connect method:

```
from playhouse.signals import connect, post_save

@connect(post_save, name='project.cache_buster')
def cache_bust_handler(sender, instance, *args, **kwargs):
    # bust the cache for this instance
    cache.delete(cache_key_for(instance))
```

1.9.5 Sqlite Extensions

The sqlite extensions module provides a number of "sqlite-only" functions, including:

- *Full-text search support*
- *Finer-grained transaction controls*
- *Custom aggregation functions, collations and user-defined functions*

Warning: In order to start using the features described below, you will need to use the extension `SqliteExtDatabase` class instead of `SqliteDatabase`.

The code below will assume you are using the following database and base model:

```
from playhouse.sqlite_ext import *

ext_db = SqliteExtDatabase('tmp.db')

class BaseExtModel(Model):
    class Meta:
        database = ext_db
```

Full-text search

Sqlite ships on most distributions with a full-text search (FTS) extension module. This can be used to expose search on your peewee models with very little work. A complete overview of sqlite's FTS is beyond the scope of this section, so please read their documentation for the details.

To use FTS with your peewee models, you must subclass the `playhouse.sqlite_ext.FTSModel`. You can store data directly in this model or you can create a separate model that references an existing model. Since virtual tables do not support column indexes, this decision will depend on how you intend to query the data stored in the full-text index.

Here is a simple example, showing the use of a separate model for storage (note that we "mix-in" the `FTSModel`):

```
class Post(BaseExtModel):
    message = TextField()

class FTSPost(Post, FTSModel):
    pass
```

When you create the table, you can specify a number of options for the full-text module, including a "source" table and a tokenizer:

```
Post.create_table()
FTSPost.create_table(content_model=Post, tokenize='porter')
```

The above code instructs sqlite to create a virtual table storing our posts that will be suitable for FTS.

```
bulk_import_some_posts()

# rebuild the search index -- this will load up the contents of the Post table
# and make it searchable via the FTSPost
FTSPost.rebuild()

# you can add/update/delete items from FTSPost just like a normal model
FTSPost.create(message='this will be searchable as well')

# perform a search
FTSPost.select().where(message__match='search phrase')
```

```
# search supports some advanced queries http://www.sqlite.org/fts3.html#section_3_1
FTSPost.select().where(message__match='cats NOT dogs')
```

There is also support for ordering search results by rank. The implementation is based on the C implementation found at the bottom of the FTS docs:

```
FTSPost.select(['*', Rank('msg_rank')]).where(message__match='python').order_by(('msg_rank', 'desc'))
```

Granular Transactions

Sqlite uses three different types of locks to control access during transactions. Details on the three types can be found in the docs, but here is a quick overview:

`deferred` locks are not acquired until the last moment. multiple processes can continue to read the database.

`immediate` lock is acquired and no further writes are possible until lock is released, but other processes can continue to read. Additionally, no other immediate or exclusive locks can be acquired.

`exclusive` lock is acquired and no further reads or writes are possible until lock is released

These various types of transactions can be opened using the special context-manager:

```
with ext_db.granular_transaction('exclusive'):
    # no other connections can read or write to the database now
    execute_some_queries()

# safe for other processes to read and write again
do_some_other_stuff()
```

Custom aggregators, collations and user-defined functions

Sqlite allows you to specify custom functions that can stand-in as aggregators, collations or functions, and then be executed as part of your queries. If you read the notes on the full-text search extension, the "sort by rank" is implemented as a user-defined function.

Python's sqlite documentation gives a good overview of how these types of functions can be used.

- custom aggregates

  ```
  class WeightedAverage(object):
      def __init__(self):
          self.total_weight = 0.0
          self.total_ct = 0.0

      def step(self, value, wt=None):
          wt = wt or 1.0
          self.total_weight += wt
          self.total_ct += wt * value

      def finalize(self):
          if self.total_weight != 0.0:
              return self.total_ct / self.total_weight
          return 0.0

  ext_db.adapter.register_aggregate(WeightedAverage, 2, 'weighted_avg')
  ```

- custom collations

```
def collate_reverse(s1, s2):
    return -cmp(s1, s2)

ext_db.adapter.register_collation(collate_reverse)
```

- custom functions

```
def sha1(s):
    return hashlib.sha1(s).hexdigest()

ext_db.adapter.register_function(sha1)
```

1.9.6 Swee'pea, syntactic sugar for peewee

Calling it syntactic sugar is a bit of a stretch. I wrote this stuff for fun after learning about ISBL from a coworker. The blog post can be found here.

At any rate, ISBL (Information Systems Base Language) is an old domain-specific language for querying relational data, developed by IBM in the 60's. Here are some example SQL and ISBL queries:

```
-- query the database for all active users
SELECT id, username, active FROM users WHERE active = True

-- query for tweets and the username of the sender
SELECT t.id, t.message, u.username
FROM tweets AS t
INNER JOIN users AS u
    ON t.user_id = u.id
WHERE u.active = True

-- tables appear first -- the colon indicates a restriction (our where clause)
-- and after the modulo is the "projection", or columns we want to select
users : active = True % (id, username, active)

(tweets * users) : user.active = True % (tweet.id, tweet.message, user.username)
```

Pretty cool. In the above examples:

- multiplication signifies a join, the tables to query (FROM)
- a colon signifies a restriction, the columns to filter (WHERE)
- modulo signifies a projection, the columns to return (SELECT)

I hacked up a small implementation on top of peewee. Since peewee does not support the ":" (colon) character as an infix operator, I used the "power" operator to signify a restriction:

```
# active users
User ** (User.active == True)

# tweets with the username of sender
(Tweet * User) ** (User.active == True) % (Tweet.id, Tweet.message, User.username)
```

To try out swee'pea, simply replace `from peewee import *` with `from playhouse.sweepea import *` and start writing wacky queries:

```
from playhouse.sweepea import *

class User(Model):
    username = CharField()
    active = BooleanField()

class Tweet(Model):
    user = ForeignKeyField(User)
    message = CharField()

# have fun!
(User * Tweet) ** (User.active == True)
```

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*