peewee Documentation

Release 2.0.0

charles leifer

April 29, 2014

- 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 some extensions, like hstore

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

See notes on *notes on upgrading and changes from 1.0*

CHAPTER 1

Contents:

1.1 Overview

peewee is a lightweight ORM written in python.

Examples:

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

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

# get tweets by editors ("<<" maps to IN)
Tweet.select().where(Tweet.user << editors)

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

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

# order users by number of tweets
User.select().annotate(Tweet).order_by(
    fn.Count(Tweet.id).desc()
)

# a similar way of expressing the same
User.select(
    User, fn.Count(Tweet.id).alias('ct')
).join(Tweet).group_by(User).order_by(R('ct desc'))

# do an atomic update
Counter.update(count=Counter.count + 1).where(
    Counter.url == request.url
)
```

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 Upgrading peewee

Peewee went from 2319 SLOC to 1666.

1.3.1 Goals for the new API

- consistent: there is one way of doing things
- expressive: things can be done that I never thought of

1.3.2 Changes from version 1.0

The biggest changes between 1.0 and 2.0 are in the syntax used for constructing queries. The first iteration of peewee I threw up on github was about 600 lines. I was passing around strings and dictionaries and as time went on and I added features, those strings turned into tuples and objects. This meant, though, that I needed code to handle all the possible ways of expressing something. Look at the code for parse_select.

I learned a valuable lesson: keep data in datastructures until the *absolute* last second.

With the benefit of hindsight and experience, I decided to rewrite and unify the API a bit. The result is a tradeoff. The newer syntax may be a bit more verbose at times, but at least it will be consistent.

Since seeing is believing, I will show some side-by-side comparisons. Let's pretend we're using the models from the cookbook, good ol' user and tweet:

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

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

Get me a list of all tweets by a user named "charlie":

```
# 1.0
Tweet.select().join(User).where(username='charlie')

# 2.0
Tweet.select().join(User).where(User.username == 'charlie')
```

Get me a list of tweets ordered by the authors username, then newest to oldest:

```
# 1.0 -- this is one where there are like 10 ways to express it
Tweet.select().join(User).order_by('username', (Tweet, 'created_date', 'desc'))

# 2.0
Tweet.select().join(User).order_by(User.username, Tweet.created_date.desc())
```

Get me a list of tweets created by users named "charlie" or "peewee herman", and which were created in the last week.

```
last_week = datetime.datetime.now() - datetime.timedelta(days=7)

# 1.0
Tweet.select().where(created_date__gt=last_week).join(User).where(
    Q(username='charlie') | Q(username='peewee herman')
)

# 2.0
Tweet.select().join(User).where((Tweet.created_date > last_week) & (
    (User.username == 'charlie') | (User.username == 'peewee herman')
))
```

Get me a list of users and when they last tweeted (if ever):

```
# 1.0
User.select({
    User: ['*'],
    Tweet: [Max('created_date', 'last_date')]
}).join(Tweet, 'LEFT OUTER').group_by(User)

# 2.0
User.select(
    User, fn.Max(Tweet.created_date).alias('last_date')
).join(Tweet, JOIN_LEFT_OUTER).group_by(User)
```

Let's do an atomic update on a counter model (you'll have to use your imagination):

```
# 1.0
Counter.update(count=F('count') + 1).where(url=request.url)
```

```
# 2.0
Counter.update(count=Counter.count + 1).where(Counter.url == request.url)
```

Let's find all the users whose username starts with 'a' or 'A':

```
# 1.0
User.select().where(R('LOWER(SUBSTR(username, 1, 1)) = %s', 'a'))

# 2.0
User.select().where(fn.Lower(fn.Substr(User.username, 1, 1)) == 'a')
```

I hope a couple things jump out at you from these examples. What I see is that the 1.0 API is sometimes a bit less verbose, but it relies on strings in many places (which may be fields, aliases, selections, join types, functions, etc). In the where clause stuff gets crazy as there are args being combined with bitwise operators ("Q" expressions) and also kwargs being used with django-style "double-underscore" lookups. The crazy thing is, there are so many different ways I could have expressed some of the above queries using peewee 1.0 that I had a hard time deciding which to even write.

The 2.0 API is hopefully more consistent. Selections, groupings, functions, joins and orderings all pretty much conform to the same API. Likewise, where and having clauses are handled the same way (in 1.0 the having clause is simply a raw string). The new `fn` object actually is a wrapper – whatever appears to the right of the dot (i.e. fn.*Lower*) – is treated as a function that can take any arbitrary parameters.

If you're feeling froggy and want to get coding, you might want to check out:

- *the cookbook*, which contains many practical examples
- *the example app documentation*, which shows how to build a simple twitter-like site
- *using "fn"*
- *the querying docs*, which contain an in-depth overview of the query apis

1.3.3 Changes in fields and columns

Well, for one, columns are gone. They were a shim that I used to hack in non-integer primary keys. I always thought the field SQL generation was one of the grosser parts of the module and even worse was the back-and-forth that happened between the field and column classes. So, columns are gone - its just fields - and they're hopefully a bit smaller and saner. I also cleaned up the primary key business. Basically it works like this:

- if you don't specify a primary key, one will be created named "id"
- if you do specify a primary key and it is a PrimaryKeyField (or subclass), it will be an automatically incrementing integer
- if you specify a primary key and it is anything else peewee assumes you are in control and will stay out of the way.

The API for specifying a non-auto-incrementing primary key changed:

```
# 1.0
class OldSchool(Model):
    uuid = PrimaryKeyField(column_class=VarCharColumn)

# 2.0
class NewSchool(Model):
    uuid = CharField(primary_key=True)
```

The kwargs for the Field constructor changed slightly, the biggest probably being that `db_index` was renamed to `index`.

1.3.4 Changes in database and adapter

In peewee 1.0 there were two classes that controlled access to the database – the Database subclass and an Adapter. The adapter's job was to say what features a database backend provided, what operations were valid, what column types were supported, and how to open a connection. The database was a bit higher-level and its main job was to execute queries and provide metadata about the database, like lists of tables, last insert id, etc.

I chose to consolidate these two classes, since inevitably they always went in pairs (e.g. SqliteDatabase/SqliteAdapter). The database class now encapsulates all this functionality.

1.3.5 How the SQL gets made

The first thing I started with is the QueryCompiler and the data structures it uses. You can see it start to take shape in my first commit. It takes the data structures from peewee and spits out SQL. It works recursively and knows about a few types of expressions:

- the query tree
- comparison statements like '==', 'IN', 'LIKE' which comprise the leaves of the tree
- expressions like addition, substraction, bitwise operations
- sql functions like `substr` and `lower`
- aggregate functions like `count` and `max`
- columns, which may be selected, joined on, grouped by, ordered by, used as parameters for functions and aggregates, etc.
- python objects to use as query parameters

At the heart of it is the `Expr` object, which is for "expression". It can be anything that can validly be translated into part of a SQL query.

Expressions can be nested, giving way to interesting possibilities like the following example I love which selects users whose username starts with "a":

```
User.select().where(fn.Substr(fn.Lower(User.username, 1, 1)) == 'a')
```

The "where" clause now contains a tree with one leaf. The leaf represents the nested function expression on the left-hand-side and the scalar value 'a' on the right hand side. Peewee will recursively evaluate the expressions on either side of the operation and generate the correct SQL.

Another aspect is that `Field` objects are also expressions, which makes it possible to write things like:

```
Employee.select().where(Employee.salary < (Employee.tenure * 1000) + 40000)
```

Note: I totally went crazy with operator overloading.

If you're interested in looking, the `QueryCompiler.parse_expr` method is where the bulk of the code lives.

1.4 Peewee Cookbook

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

Examples will use the following models:

```
from peewee import *

class User(Model):
    username = CharField()

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

1.4.1 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 = SqliteDatabase('custom.db')

>>> class CustomModel(Model):
...     whatev = 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 = SqliteDatabase('custom.db')

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

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

```
class Tweet(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 = PostgresqlDatabase('my_database', user='code')


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

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

Using with MySQL

Point models at an instance of `MySQLDatabase`.

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


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

class User(MySQLModel):
    username = 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` (*recommended*):

```
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. Pysqlite can share a connection across threads, so if you would prefer to reuse a connection in multiple threads:

```
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 = SqliteDatabase(None)

class SomeModel(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)
```

1.4.2 Creating, Reading, Updating and Deleting

Creating a new record

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

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

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

Alternatively, you can build up a model instance programmatically and then save it:

```
>>> user = User()
>>> user.username = 'Charlie'
>>> user.save()
>>> user.id
1
```

See also `Model.save()`, `Model.insert()` and `InsertQuery`

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`:

```
>>> user.save()
>>> user.id
1
>>> user.save()
>>> user.id
1
```

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 = Tweet.update(is_published=True).where(Tweet.creation_date < 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:

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

>>> User.get(User.id == 1)
UserDoesNotExist: instance matching query does not exist:
SQL: SELECT t1."id", t1."username" FROM "user" AS t1 WHERE t1."id" = ?
PARAMS: [1]
```

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

```
>>> delete_query = Tweet.delete().where(Tweet.pub_date < one_year_ago)
>>> delete_query.execute()
7 # <--- number of rows 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.

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.

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

>>> User.get(User.id == 1).username
u'Charlie'

>>> User.get(User.username == 'Charlie')
<__main__.Blog object at 0x2529410>

>>> User.get(User.username == 'nobody')
UserDoesNotExist: instance matching query does not exist:
SQL: SELECT t1."id", t1."username" FROM "user" AS t1 WHERE t1."username" = ?
PARAMS: ['nobody']
```

For more 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 user in User.select():
...     print user.username
...
Charlie
Peewee Herman
```

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 Users and Tweets. Tweet has a foreign key to User, meaning that any given user may have 0..n tweets. A user's related tweets are exposed using a `SelectQuery`, and can be iterated the same as any other SelectQuery:

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

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

```
>>> for tweet in user.tweets.order_by(Tweet.created_date.desc()):
...     print tweet.message
...
look at this picture of my food
this is fun
hello world
```

Filtering records

You can filter for particular records using normal python operators.

```
>>> user = User.get(User.username == 'Charlie')
>>> for tweet in Tweet.select().where(Tweet.user == user, Tweet.is_published == True):
...     print '%s: %s (%s)' % (tweet.user.username, tweet.message)
...
Charlie: hello world
Charlie: this is fun

>>> for tweet in Tweet.select().where(Tweet.created_date < datetime.datetime(2011, 1, 1)):
...     print tweet.message, tweet.created_date
...
Really old tweet 2010-01-01 00:00:00
```

You can also filter across joins:

```
>>> for tweet in Tweet.select().join(User).where(User.username == 'Charlie'):
...     print tweet.message
hello world
this is fun
look at this picture of my food
```

If you want to express a complex query, use parentheses and python's "or" and "and" operators:

```
>>> Tweet.select().join(User).where(
...     (User.username == 'Charlie') |
```

```
...     (User.username == 'Peewee Herman')
... )
```

Check out *the table of query operations* to see what types of queries are possible.

Note: A lot of fun things can go in the where clause of a query, such as:

- a field expression, e.g. `User.username == 'Charlie'`
- a function expression, e.g. `fn.Lower(fn.Substr(User.username, 1, 1)) == 'a'`
- a comparison of one column to another, e.g. `Employee.salary < (Employee.tenure * 1000) + 40000`

You can also nest queries, for example tweets by users whose username starts with "a":

```
# the "<<" operator signifies an "IN" query
Tweet.select().where(
    Tweet.user << User.select().where(fn.Lower(fn.Substr(User.username, 1, 1)) == 'a')
)
```

Note: If you are already familiar with Django's ORM, you can use the "double underscore" syntax using the `SelectQuery.filter()` method:

```
>>> for tweet in Tweet.filter(user__username='Charlie'):
...     print tweet.message
hello world
this is fun
look at this picture of my food
```

To perform OR lookups, use the special `DQ` object:

```
>>> User.filter(DQ(username='Charlie') | DQ(username='Peewee Herman'))
```

Warning: The *Zen of Python* says "There should be one– and preferably only one –obvious way to do it." The django-style filtering is supported for backwards compatibility with 1.0, so if you can, its probably best not to use it.

Check *the docs* for some more example queries.

Sorting records

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

>>> for t in Tweet.select().order_by(Tweet.created_date.desc()):
...     print t.pub_date
...
2011-06-07 14:12:57
2011-06-07 14:08:48
2010-01-01 00:00:00
```

You can also order across joins. Assuming you want to order tweets by the username of the author, then by created_date:

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

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 tweet in Tweet.select().order_by(Tweet.id).paginate(2, 10):
...     print tweet.message
...
tweet 10
tweet 11
tweet 12
tweet 13
tweet 14
tweet 15
tweet 16
tweet 17
tweet 18
tweet 19
```

Counting records

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

```
>>> Tweet.select().count()
100
>>> Tweet.select().where(Tweet.id > 50).count()
50
```

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

```
>>> Stat.update(counter=Stat.counter + 1).where(Stat.url == request.url)
```

Aggregating records

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

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

This is equivalent to the following:

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

The resulting query will return User objects with all their normal attributes plus an additional attribute 'count' which will contain the number of tweets. 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 users with 0 tweets:

```
query = User.select().join(Tweet, JOIN_LEFT_OUTER).annotate(Tweet)
```

You can also specify a custom aggregator:

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

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(fn.Count(Photo.id) > 5)
```

Yields the following:

```
SELECT t1."id", t1."name"
FROM "tag" AS t1
INNER JOIN "phototag" AS t2 ON t1."id" = t2."tag_id"
INNER JOIN "photo" AS t3 ON t2."photo_id" = t3."id"
GROUP BY t1."id", t1."name"
HAVING Count(t3."id") > 5
```

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

```
>>> Tag.select(
...     Tag, fn.Count(Photo.id).alias('count')
... ).join(PhotoTag).join(Photo).group_by(Tag).having(fn.Count(Photo.id) > 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 `fn()` function to construct queries:

```
# select the users' id, username and the first letter of their username, lower-cased
query = User.select(User, fn.Lower(fn.Substr(User.username, 1, 1)).alias('first_letter'))

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

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

There are times when you may want to simply pass in some arbitrary sql. You can do this using the special `R` class. One use-case is when referencing an alias:

```
# we'll query the user table and annotate it with a count of tweets for
# the given user
query = User.select(User, fn.Count(Tweet.id).alias('ct')).join(Tweet).group_by(User)

# now we will order by the count, which was aliased to "ct"
query = query.order_by(R('ct'))
```

1.4.3 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():
    user.delete_instance(recursive=True) # delete user and associated tweets
```

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_user(user):
    user.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:
    user.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)

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

1.4.4 Non-integer Primary Keys and other Tricks

Non-integer primary keys

If you would like use a non-integer primary key (which I generally don't recommend), you can override the default `column_class` of the `PrimaryKeyField`:

```
from peewee import *

class UUIDModel(Model):
    id = CharField(primary_key=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 storage type as the primary key they are related to.

See full documentation on *non-integer primary keys*.

Bulk loading data or manually specifying primary keys

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

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

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

User._meta.auto_increment = True
```

If you *always* want to have control over the primary key, simply do not use the `PrimaryKeyField` type:

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

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

1.4.5 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:

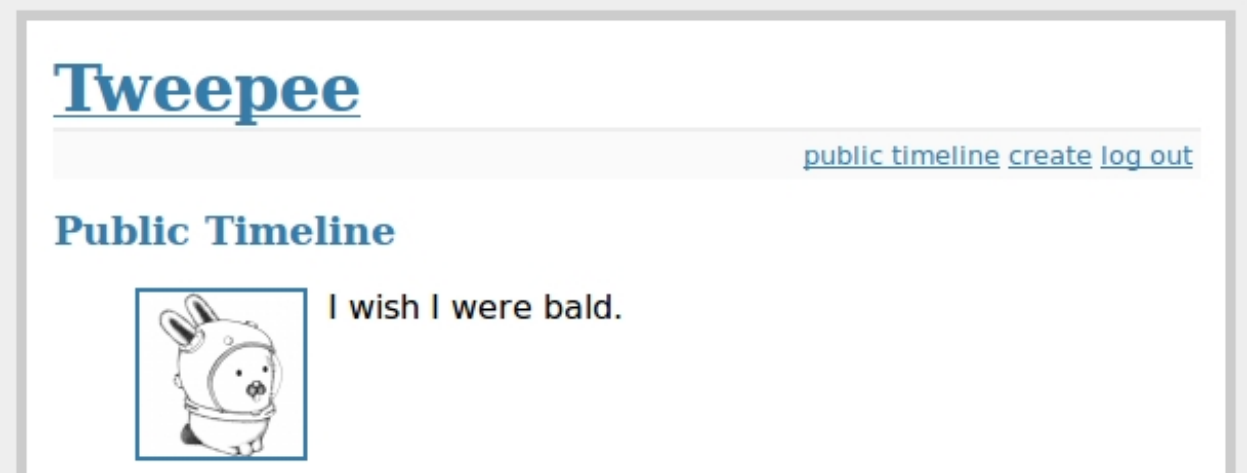
- database connection object
- a base model class to use this connection
- models that were introspected from the database tables

The generated code is written to stdout.

1.4.6 Schema migrations

Currently peewee does not have support for automatic schema migrations.

1.5 Example app



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
```

1.5.1 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
```

1.5.2 Diving into the code

Models

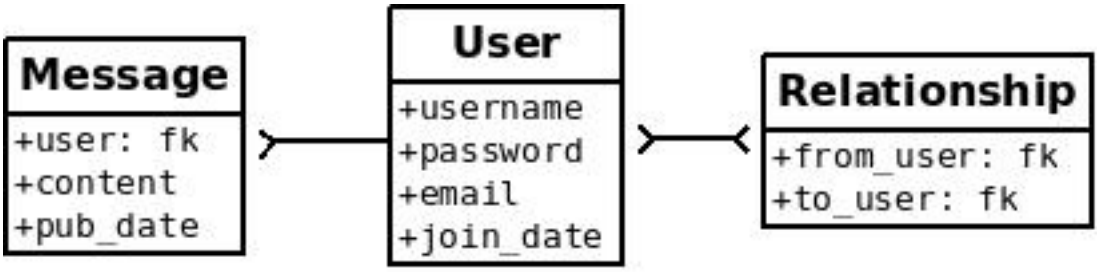
In the spirit of the ur-python framework, django, peewee uses declarative model definitions. If you're not familiar with django, the idea is that you declare a class with some members which map directly to the database schema. For the twitter clone, there are just three models:

`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:

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

# model definitions -- the standard "pattern" is to define a base model class
# that specifies which database to use.  then, any subclasses will automatically
# use the correct storage. for more information, see:
# http://charlesleifer.com/docs/peewee/peewee/models.html#model-api-smells-like-django
class BaseModel(Model):
    class Meta:
        database = database

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

    class Meta:
        order_by = ('username',)

    # it often makes sense to put convenience methods on model instances, for
    # example, "give me all the users this user is following":
    def following(self):
        # query other users through the "relationship" table
        return User.select().join(
            Relationship, on=Relationship.to_user,
        ).where(Relationship.from_user == self)

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

    def is_following(self, user):
        return Relationship.select().where(
            (Relationship.from_user == self) &
            (Relationship.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)


# this model contains two foreign keys to user -- it essentially allows us to
```

```
# model a "many-to-many" relationship between users.  by querying and joining
# on different columns we can expose who a user is "related to" and who is
# "related to" a given user
class Relationship(BaseModel):
    from_user = ForeignKeyField(User, related_name='relationships')
    to_user = ForeignKeyField(User, related_name='related_to')


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

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

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

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

The queries end up looking like:

```
# following:
SELECT t1."id", t1."username", t1."password", t1."email", t1."join_date"
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 t1."username" ASC

# followers
SELECT t1."id", t1."username", t1."password", t1."email", t1."join_date"
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 t1."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()`.

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

Results in the following SQL query:

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

1.6 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 User(BaseModel):
    username = CharField()

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

```
    created_date = DateTimeField(default=datetime.datetime.now)
    is_published = BooleanField(default=True)
```

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 User(BaseModel):
       username = 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.

1.6.1 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
User.create_table()
Tweet.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.

1.6.2 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:

```
>>> user = User.create(username='charlie')
>>> tweet = Tweet.create(
...     message='http://www.youtube.com/watch?v=xdhLQCYQ-nQ',
...     user=user
... )

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

2. Build up the instance programmatically:

```
>>> user = User()
>>> user.username = 'charlie'
>>> user.save()
```

Traversing foriegn keys

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

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

The reverse is also true, we can iterate a `User` objects associated `Tweets`:

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

Under the hood, the `tweets` attribute is just a `SelectQuery` with the where clause prepopulated to point at the right `User` instance:

```
>>> user.tweets
<peewee.SelectQuery object at 0x151f510>
```

1.6.3 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
- order_by: a sequence of columns to use as the default ordering for this model

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 Tweet(Model):
    message = TextField()
    created = DateTimeField()

    class Meta:
        # order by created date descending
        ordering = ('-created',)
```

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
```

1.6.4 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_nullable=False*]])

Parameters

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

Delete the given instance. Any foreign keys set to cascade on delete will be deleted automatically. For more programmatic control, you can call with recursive=True, which will delete any non-nullable related models (those that *are* nullable will be set to NULL). If you wish to delete all dependencies regardless of whether they are nullable, set `delete_nullable=True`.

example:

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

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

Parameters

- **args** – a list of query expressions, e.g. `Usre.username == 'foo'`
- **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(User.username == username, User.password == password)
```

This method is also expose via the `SelectQuery`, though it takes no parameters:

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

Note: the "kwargs" style syntax is provided for compatibility with version 1.0. The expression-style syntax is preferable.

---

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`**(**selection*)

Parameters **selection** – a list of model classes, field instances, functions or expressions

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

example:

```
>>> User.select().where(User.active == True).order_by(User.username)
>>> Tweet.select(Tweet, User).join(User).order_by(Tweet.created_date.desc())
```

classmethod **`update`**(***query*)

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

example:

```
>>> q = User.update(active=False).where(User.registration_expired == True)
>>> q.execute() # <-- execute it
```

classmethod **`delete`**()

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

example:

```
>>> q = User.delete().where(User.active == False)
>>> 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.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 **`filter`**(**args*, ***kwargs*)

Parameters

- **args** – a list of `DQ` 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')
```

Note: this method is provided for compatibility with peewee 1.0

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

1.7 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
```

1.7.1 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
- `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`
- `primary_key = False` – whether this field is the primary key for the table
- `sequence = None` – sequence to populate field (if backend supports it)

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
ForeignKeyField	rel_model, 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.

1.7.2 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)
```

1.7.3 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(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(Student.name == 'da vinci'):
    print course.name
```

To efficiently iterate over a many-to-many relation, i.e., list all students and their respective courses, we will query the "through" model `StudentCourse` and "precompute" the Student and Course:

```
query = StudentCourse.select(
    StudentCourse, Student, Course)
).join(Course).switch(StudentCourse).join(Student)
```

To print a list of students and their courses you might do the following:

```
last = None
for student_course in query:
    student = student_course.student
    if student != last:
        last = student
        print 'Student: %s' % student.name
    print '    - %s' % student_course.course.name
```

Since we selected all fields from `Student` and `Course` in the `select` clause of the query, these foreign key traversals are "free" and we've done the whole iteration with just 1 query.

1.7.4 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 = CharField(primary_key=True)
```

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 value 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 storage type as the primary key they are related to.

1.7.5 Field class API

class `Field`

The base class from which all other field types extend.

`db_field = '<some field type>'`

Attribute used to map this field to a column type, e.g. "string" or "datetime"

template = '%(column_type)s'
A template for generating the SQL for this field

__init__(*null=False*, *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
- **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`
- **primary_key** (*boolean*) – whether to use this as the primary key for the table
- **sequence** – name of sequence (if backend supports it)

db_value(*value*)

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

Return type converted python datatype

python_value(*value*)

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

Return type python data type

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

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

Return type python data type

field_attributes()
This method is responsible for return a dictionary containing the default field attributes for the column, e.g. `{'max_length': 255}`

Return type a python dictionary

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.

class ForeignKeyField
Stores: relationship to another model

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

Parameters

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

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

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

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

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

1.8 Querying API

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

Looking at some simple queries

Get active users:

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

Get users who are either staff or superusers:

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

Get tweets by user named "charlie":

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

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

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

1.8.2 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(User.is_staff == True)
```

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

Joining

You can join on tables related to one another by `ForeignKeyField`. The `join()` 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

There are three types of joins by default:

- JOIN_INNER (default)
- JOIN_LEFT_OUTER
- JOIN_FULL

Here is an example using JOINs:

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

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

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

Another way to write the above query would be to use a subquery:

```
>>> staff = User.select().where(User.is_staff == true)
>>> Blog.select().where(Blog.status == LIVE, Blog.user << staff)
```

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:

```
-- translates roughly to --
SELECT t1.* FROM blog AS t1
WHERE (
    t1.status = ? AND
    t1.user_id IN (
        SELECT t2.id FROM user AS t2 WHERE t2.is_staff = ?
```

```
    )
)
```

And here it is using joins:

```
-- and here it would be if 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:

Lookup	Meaning
==	x equals y
<	x is less than y
<=	x is less than or equal to y
>	x is greater than y
>=	x is greater than or equal to y
!=	x is not equal to y
<<	x IN y, where y is a list or query
>>	x IS y, where y is None/NULL
%	x LIKE y where y may contain wildcards
**	x ILIKE y where y may contain wildcards

1.8.3 Performing advanced queries

As you may have noticed, all the examples up to now have shown queries that combine multiple clauses with "AND". To create arbitrarily complex queries, simply use python's bitwise "and" and "or" operators:

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

The `WHERE` clause will look something like:

```
WHERE (is_staff = ? OR is_superuser = ?)
```

In order to negate an expression, use the bitwise "invert" operator:

```
>>> staff_users = User.select().where(is_staff=True)
>>> Tweet.select().where(
...     ~(Tweet.user << staff_users)
... )
```

This query generates roughly the following SQL:

```
SELECT t1.* FROM blog AS t1
WHERE
    NOT t1.user_id IN (
        SELECT t2.id FROM user AS t2 WHERE t2.is_staff = ?
    )
```

Rather complex lookups are possible:

```
>>> sq = User.select().where(
...     ((User.is_staff == True) | (User.is_superuser == True)) &
...     (User.join_date >= datetime(2009, 1, 1)
... )
```

This generates roughly the following SQL:

```
SELECT * FROM user
WHERE (
    (is_staff = ? OR is_superuser = ?) AND
    (join_date >= ?)
)
```

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()
    tenure = 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)?

```
WorkerProfile.select().where(
    WorkerProfile.salary < WorkerProfile.desired
)
```

We can also create expressions, like to find employees who might not be getting paid enough based on their tenure:

```
WorkerProfile.select().where(
    WorkerProfile.salary < (WorkerProfile.tenure * 1000) + 40000
)
```

Atomic updates

The techniques shown above also work for updating data. Suppose you are counting pageviews in a special table:

```
PageView.update(count=PageView.count + 1).where(
    PageView.url == request.url
)
```

The "fn" helper

class fn
A helper class that will convert arbitrary function calls to SQL function calls.

SQL provides a number of helper functions as a part of the language. These functions can be used to calculate counts and sums over rows, perform string manipulations, do complex math, and more. There are a lot of functions.

To express functions in peewee, use the `fn` object. The way it works is anything to the right of the "dot" operator will be treated as a function. You can pass that function arbitrary parameters which can be other valid expressions.

For example:

Peewee expression	Equivalent SQL
`fn.Count(Tweet.id).alias('count')`	`Count(t1."id") AS count`
`fn.Lower(fn.Substr(User.username, 1, 1))`	`Lower(Substr(t1."username", 1, 1))`
`fn.Rand().alias('random')`	`Rand() AS random`
`fn.Stddev(Employee.salary).alias('sdv')`	`Stddev(t1."salary") AS sdv`

Functions can be used as any part of a query:

- select
- where
- group_by
- order_by
- having
- update query
- insert query

Aggregating records

Suppose you have some users and want to get a list of them along with the count of tweets each has made. First I will show you the shortcut:

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

This is equivalent to the following:

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

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

```
query = User.select().join(Tweet, JOIN_LEFT_OUTER).annotate(Tweet)
```

You can also specify a custom aggregator. In the following query we will annotate the users with the date of their most recent tweet:

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

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:

```
most_recent_tweet = Tweet.select().aggregate(fn.Max(Tweet.created_date))
```

SQL Functions

Arbitrary SQL functions can be expressed using the `fn` function.

Selecting users and counts of tweets:

```
>>> users = User.select(User, fn.Count(Tweet.id).alias('count')).join(Tweet).group_by(User)
>>> for user in users:
...     print user.username, 'posted', user.count, 'tweets'
```

This functionality can also be used as part of the `WHERE` or `HAVING` clauses:

```
>>> a_users = User.select().where(fn.Lower(fn.Substr(User.username, 1, 1)) == 'a')
>>> for user in a_users:
...    print user.username

alpha
Alton
```

Saving Queries by Selecting Related Models

Returning to my favorite models, `User` and `Tweet`, between which there is a `ForeignKeyField`, a common pattern might be to display a list of the latest 10 tweets with some info about the user that posted them. We can do this pretty easily:

```
for tweet in Tweet.select().order_by(Tweet.created_date.desc()).limit(10):
    print '%s, posted on %s' % (tweet.message, tweet.user.username)
```

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

- 1 query for the tweets
- 1 query for every related user (10 total)

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

```
tweets = Tweet.select(Tweet, User).join(User)
for tweet in tweets.order_by(Tweet.created_date.desc()).limit(10):
    print '%s, posted on %s' % (tweet.message, tweet.user.username)
```

Will cause only one query that looks something like this:

```
SELECT t1.id, t1.message, t1.user_id, t1.created_date, t2.id, t2.username
FROM tweet AS t1
INNER JOIN user AS t2
    ON t1.user_id = t2.id
ORDER BY t1.created_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(User)`

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 users and prefetch all related tweets.

This *can* be done by fetching all tweets (with related user data), then reconstructing the users 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
tweets = Tweet.select(Tweet, User).join(User)
for tweet in tweets.order_by(Tweet.created_date.desc()).limit(10):
    print '%s, posted on %s' % (tweet.message, tweet.user.username)
```

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"
tweets = Tweet.select(Tweet, User.username).join(User).naive()
for tweet in tweets.order_by(Tweet.created_date.desc()).limit(10):
    print '%s, posted on %s' % (tweet.message, tweet.username)
```

1.8.4 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(User.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(User.id > 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
8

>>> sq = User.select().where(User.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.

1.8.5 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.

1.8.6 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*, **selection*)

Parameters

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

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

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

`where`(**q_or_node*)

Parameters **q_or_node** – a list of expressions (`Q` or `Node` objects

Return type a `SelectQuery` instance

```
>>> sq = SelectQuery(User).where(User.username == 'somebody')
>>> sq = SelectQuery(Blog).where(
...     (User.username == 'somebody') |
...     (User.username == 'nobody')
... )
```

Note: `where()` calls are chainable

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

Parameters

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

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

group_by(**clauses*)

Parameters **clauses** – either a list of model classes or field names

Return type `SelectQuery`

```
>>> # get a list of blogs with the count of entries each has
>>> sq = User.select(
...     User, fn.Count(Tweet.id).alias('count')
... ).join(Tweet).group_by(User)
```

having(**q_or_node*)

Parameters **q_or_node** – a list of expressions (`Q` or `Node` objects

Return type `SelectQuery`

```
>>> sq = User.select(
...     User, fn.Count(Tweet.id).alias('count')
... ).join(Tweet).group_by(User).having(fn.Count(Tweet.id) > 10)
```

order_by(**clauses*)

Parameters **clauses** – a list of fields or calls to `field.[asc|desc]()`

Return type `SelectQuery`

example:

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

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.

```
>>> User.select().order_by(User.username).paginate(3, 20) # <-- get users 41-60
```

limit(*num*)

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

offset(*num*)

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

count()

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

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

get()

Return type `Model` instance or raises `DoesNotExist` exception

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

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

This method is also exposed via the `Model` api, in which case it accepts arguments that are translated to the where clause:

```
>>> user = User.get(User.active == True, User.username == username)
```

exists()

Return type boolean whether the current query will return any rows. uses an optimized lookup, so use this rather than `get()`.

```
>>> sq = User.select().where(User.active == True)
>>> if sq.where(User.username==username, User.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. `fn.Count(Tweet.id).alias('count')`

Return type `SelectQuery`

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

```
>>> User.select().annotate(Tweet)
```

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

```
>>> user_latest = User.select().annotate(Tweet, fn.Max(Tweet.created_date).alias('latest'))
```

Note: If the `ForeignKeyField` is `nullable`, then a `LEFT OUTER` join may need to be used:

```
>>> User.select().join(Tweet, JOIN_LEFT_OUTER).annotate(Tweet)
```

---

aggregate(*aggregation*)

Parameters **aggregation** – a function specifying what aggregation to perform, for example `fn.Max(Tweet.created_date)`.

Method to look at an aggregate of rows using a given function and return a scalar value, such as the count of all rows or the average value of a particular column.

for_update([*for_update=True*])

Return type `SelectQuery`

indicates that this query should lock rows for update

distinct()

Return type `SelectQuery`

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

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

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

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

Parameters

- **args** – a list of `DQ` 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')
```

Note: this method is provided for compatibility with peewee 1.

---

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(User.active == True):
...     print user.username
```

1.8.7 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(User.registration_expired==True)
>>> uq.execute() # run the query

>>> atomic_update = UpdateQuery(User, message_count=User.message_count + 1).where(User.id ==
>>> atomic_update.execute() # run the query
```

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

Same as `SelectQuery.where()`

execute()

Return type Number of rows updated

Performs the query

1.8.8 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(User.active==False)
```

where(**args*, ***kwargs*)
Same as `SelectQuery.where()`

execute()

Return type Number of rows deleted

Performs the query

1.8.9 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)
>>> iq.execute() # <--- insert new row
```

execute()

Return type primary key of the new row

Performs the query

1.8.10 RawQuery

class RawQuery
Allows execution of an arbitrary 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

1.9 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` encapsulates functionality specific to a given db driver. For example difference in column types across database engines, or support for certain features like sequences. The database is responsible for smoothing out the quirks of each backend driver to provide a consistent interface.

The `Database` also uses a subclass of `QueryCompiler` to generate valid SQL. The QueryCompiler maps the internal data structures used by peewee to SQL statements.

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 `QueryCompiler` will be of interest to anyone interested in adding support for another database driver.

1.9.1 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*

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

Starting out

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

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


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

Essential methods to override

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

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

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

    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
```

Other things the database 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
- mapping operations such as "LIKE/ILIKE" to their database equivalent

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()
```

1.9.2 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

compiler_class = QueryCompiler
A class suitable for compiling queries

expr_overrides = {}
A mapping of expression codes to string operators

field_overrides = {}
A mapping of field types to database column types, e.g. `{'primary_key': 'SERIAL'}`

for_update = False
Whether the given backend supports selecting rows for update

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

op_overrides = {}
A mapping of operation codes to string operations, e.g. `{OP_LIKE: 'LIKE BINARY'}`

quote_char = '"'
The string used by the driver to quote names

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

sequences = False
Whether the given backend supports sequences

subquery_delete_same_table = True
Whether the given backend supports deleting rows using a subquery that selects from the same table

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

Parameters

- **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

get_compiler()

Return type an instance of `QueryCompiler`

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(*query*)

Param a query instance, such as a `SelectQuery`

Return type the resulting cursor

execute_sql(*sql*[, *params=None*[, *require_commit=True*]])

Parameters

- **sql** – a string sql query
- **params** – a list or tuple of parameters to interpolate

Note: You can configure whether queries will automatically commit by using the `set_autocommit()` and `Database.get_autocommit()` methods.

---

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*)

Parameters **model_class** – `Model` class to create table for

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

Parameters

- **model_class** – `Model` table on which to create index
- **fields** – field(s) to create index on (either field instances or field names)
- **unique** – whether the index should enforce uniqueness

create_foreign_key(*model_class*, *field*)

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.

---

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

1.10 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.

1.10.1 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

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

1.10.2 Postgresql HStore

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

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

You can filter by keys or partial dictionary:

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

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

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

Or, alternatively an atomic delete:

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

Multiple keys can be deleted at the same time:

```
>>> query = House.update(features=f.delete('garage', 'sqft'))
```

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

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

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

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

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

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

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

You can retrieve a slice of data, for example, all the garage data:

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

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

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

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

123 Main St True

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

123 Main St 2 cars
```

1.10.3 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

1.10.4 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))
```

CHAPTER 2

Indices and tables

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