# peewee-moves

*Release 2.1.0*

**Oct 21, 2020**

# Contents

A simple and flexible migration manager for Peewee ORM.

CHAPTER 1

# Requirements

- python >= 3.4, <= 3.6
- peewee >= 3.0.0

CHAPTER 2

# Installation

This package can be installed using pip:

```
pip install peewee-moves
```

# CHAPTER 3

# Usage

Here's a quick teaser of what you can do with peewee-moves:

```
$ export FLASK_APP=myflaskapp

$ flask db create app.models.Category
INFO: created migration 0001_create_table_category

$ flask db revision "do something"
INFO: created migration 0002_do_something

$ flask db upgrade
INFO: 0001_create_table_category: upgrade
INFO: 0002_do_something: upgrade

$ flask db downgrade
INFO: 0002_do_something: downgrade

$ flask db status
INFO: 0001_create_table_category: applied
INFO: 0002_do_something: pending
```

And if you're curious, here's what *0001_create_table_category.py* looks like. A migration was automatically created based on the definition of the Category model.

```
def upgrade(migrator):
    with migrator.create_table('category') as table:
        table.primary_key('id')
        table.integer('code', unique=True)
        table.string('name', max_length=250)

def downgrade(migrator):
    migrator.drop_table('category')
```

CHAPTER 4

# Documentation

Check out the Full Documentation for more details.

CHAPTER 5

# Contents

## 5.1 Direct Usage

The main entry point is the *DatabaseManager* class which accepts the following parameters:

- database: location of database (URL string, dictionary, or `peewee.Database` instance).
- table_name: table name to store migration history (default: migration_history)
- directory: directory to store migration files (default: migrations)

```
from peewee import SqliteDatabase
from peewee_moves import DatabaseManager

manager = DatabaseManager(SqliteDatabase('test.sqlite'))
manager = DatabaseManager({'engine': 'peewee.SqliteDatabase', 'name': 'test.sqlite'})
manager = DatabaseManager('sqlite:///test.sqlite')
```

From there, you can call methods to manage and run migration files.

### 5.1.1 New Migration

This will create a blank migration file with the next ID and the default name of "auto migration" or whatever name you provide

```
>>> manager.revision()
created: INFO: 0001_auto_migration

>>> manager.revision('custom name')
created: INFO: 0002_custom_name
```

### 5.1.2 Database Upgrade

This will run the upgrade() method in each unapplied migration. If you specify a target, the migrator will only run upgrades through that target. If no target is specified, all unapplied migrations will run.

```
>>> manager.upgrade('0001')
INFO: upgrade: 0001_auto_migration

>>> manager.upgrade()
INFO: upgrade: 0002_custom_name
INFO: upgrade: 0003_another_migration
```

Pass the *–fake* flag to add the record to the migration history table but don't actually run the migration:

```
>>> manager.upgrade(fake=True)
INFO: upgrade: 0003_another_migration

>>> manager.upgrade('0001', fake=True)
INFO: upgrade: 0002_custom_name
INFO: upgrade: 0001_auto_migration
```

### 5.1.3 Database Downgrade

This does the opposite of upgrade(). It calls the downgrade() method on each applied migration. If you specify a target, the migrator will only run downgrades through that target. If no target is specified, only the most recent migration will be downgraded.

```
>>> manager.downgrade()
INFO: downgrade: 0003_another_migration

>>> manager.downgrade('0001')
INFO: downgrade: 0002_custom_name
INFO: downgrade: 0001_auto_migration
```

Pass the *–fake* flag to remove the record from migration history table but don't actually run the migration:

```
>>> manager.downgrade(fake=True)
INFO: downgrade: 0003_another_migration

>>> manager.downgrade('0001', fake=True)
INFO: downgrade: 0002_custom_name
INFO: downgrade: 0001_auto_migration
```

### 5.1.4 Delete Migration

This will remove a migration from the database and the filesystem, as if it never happened. You might never need this, but it could be useful in some circumstances.

```
>>> manager.delete('0003')
INFO: deleted: 0003_another_migration
```

### 5.1.5 Migration Status

This will simply show the status of each migration file so you can see which ones have been applied.

```
>>> manager.status()
INFO: [x] 0001_auto_migration
INFO: [x] 0002_custom_name
INFO: [ ] 0003_another_migration
```

### 5.1.6 Automagic Migration Creation

It's possible to create a migration file automatically that will have the operations necessary to upgrade and downgrade your existing models.

Let's say you have the following two models defined in *models.py*:

```
import peewee

class Group(peewee.Model):
    code = peewee.IntegerField(unique=True)
    name = peewee.CharField(max_length=250)

    class Meta:
        db_table = 'auth_groups'

class User(peewee.Model):
    name = peewee.CharField(max_length=250)
    group = peewee.ForeignKeyField(Group, related_name='users')

    class Meta:
        db_table = 'auth_users'
        indexes = (
            (('name', 'group'), True),
        )
```

Running the following command will create the migration file necessary to upgrade/downgrade the Group model.

```
>>> migrator.create('models.Group')
INFO: created: 0001_create_table_auth_groups
```

You can also pass a module to create migration files for all models within:

```
>>> migrator.create('models')
INFO: created: 0001_create_table_auth_groups
INFO: created: 0002_create_table_auth_users
```

Let's look at both those files:

**0001_create_table_auth_groups.py**

```
def upgrade(migrator):
    with migrator.create_table('auth_groups') as table:
        table.primary_key('id')
        table.int('code', unique=True)
        table.char('name', max_length=250)
```

(continues on next page)

(continued from previous page)

```
def downgrade(migrator):
    migrator.drop_table('auth_groups')
```

**0002_create_table_auth_users.py**

```
def upgrade(migrator):
    with migrator.create_table('auth_users') as table:
        table.primary_key('id')
        table.char('name', max_length=250)
        table.foreign_key('int', 'group_id', references='auth_groups.id')
        table.add_index(('name', 'group_id'), unique=True)

def downgrade(migrator):
    migrator.drop_table('auth_users')
```

As you can see, this creates all the operations necessary to create the table for both models.

The user model has a foreign key to the groups model, but the migration file for users does not contain a dependency on the Group model! This is intentional. If the Group model changes or gets removed in a future migration, this migration will not be impacted and can still run any time a new database needs to be set up.

This currently only supports creating models. If your model changes, it's up to you to write the migration to support that.

## 5.2 Migrator API

The previous exmple shows the files that were created automatically to support two models. The argument to upgrade() and downgrade() is a migrator instance that has a database-agnostic API. This allows you to write command in Python that will get executed against the database when upgrade() and downgrade() are called.

Here's a full example of everything you can do in either upgrade() or downgrade() using the migrator API:

```
with migrator.create_table(name, safe=False) as table:
    table.primary_key('colname', **kwargs)
    table.bare('colname', **kwargs)
    table.biginteger('colname', **kwargs)
    table.binary('colname', **kwargs)
    table.blob('colname', **kwargs)
    table.bool('colname', **kwargs)
    table.date('colname', **kwargs)
    table.datetime('colname', **kwargs)
    table.decimal('colname', **kwargs)
    table.double('colname', **kwargs)
    table.fixed('colname', **kwargs)
    table.float('colname', **kwargs)
    table.integer('colname', **kwargs)
    table.char('colname', **kwargs)
    table.text('colname', **kwargs)
    table.time('colname', **kwargs)
    table.uuid('colname', **kwargs)
    table.foreign_key('coltype', 'colname', references='othertable.col')
    table.add_index(('col1', 'col2'), unique=True)
    table.add_constraint('constraint string')

migrator.drop_table('name', safe=False, cascade=False)
```

(continues on next page)

(continued from previous page)

```
migrator.add_column('table', 'name', 'type', **kwargs)
migrator.drop_column('table', 'name', 'field', cascade=True)
migrator.rename_column('table', 'old_name', 'new_name')
migrator.rename('table', 'old_name', 'new_name')
migrator.add_not_null('table', 'column')
migrator.drop_not_null('table', 'column')
migrator.add_index('table', 'columns', unique=False)
migrator.drop_index('table', 'index_name')
cursor = migrator.execute_sql(sql, params=None)
```

The kwargs are passed to the field as they would be if you were defining the field on a Peewee model class.

The migrator.execute_sql allows for writing direct SQL if you need to. There's nothing stopping you from writing something specific to your database engine using this method.

And remember, the migration files are just Python! So you can import and run other Python code if needed.

## 5.3 Command Line Usage

A command named `peewee-db` is automatically installed with this package. This command allows you to easily issue database management commands without using the Python API directly:

```
$ peewee-db --help

Usage: peewee-db [OPTIONS] COMMAND [ARGS]...

    Run database migration commands.

Options:
    --directory TEXT  [required]
    --database TEXT   [required]
    --table TEXT
    --help            Show this message and exit.

Commands:
    create     Create a migration based on an existing...
    delete     Delete the target migration from the...
    downgrade  Run database downgrades.
    info       Show information about the current database.
    revision   Create a blank migration file.
    status     Show information about migration status.
    upgrade    Run database upgrades.
```

Each command requires that you specify a `database` and `directory` where `database` is the URL to your database and `directory` is where migration files are stored.

For example, here's how you can show the status of your database:

```
$ peewee-db --database=sqlite:///mydata.sqlite --directory=migrations status
INFO: [ ] 0001_create_table_auth_groups
INFO: [ ] 0002_create_table_auth_users
```

And to create a new revision file you can do this:

```
$ peewee-db --database=sqlite:///mydata.sqlite --directory=migrations revision
↪"custom revision"
INFO: created: 0003_custom_revision
```

Pass the *–fake* flag to *upgrade* or *downgrade* to update the migration history table without running the migration:

## 5.4 Flask Usage

This package includes an interface to Flask versions 0.11 or later using Click which provides an easy-to-use command line interface. If you are using Flask 0.10, you can use backported integration via Flask-CLI.

For this to work properly, you must define a configuration variable named `DATABASE` in your Flask app config:

```
app = Flask(__name__)
app.config['DATABASE'] = 'sqlite:///database.sqlite'
```

This can be a connection string as shown above, or also a dict or `peewee.Database` instance.

```
app.config['DATABASE'] = SqliteDatabase('test.sqlite')

app.config['DATABASE'] = {
    'engine': 'peewee.SqliteDatabase',
    'name': 'test.sqlite'
}
```

The `db` command will automatically add the command to the cli if Flask is installed:

```
flask db --help
```

This gives you the following command line interface:

```
$ flask db --help
Usage: flask db [OPTIONS] COMMAND [ARGS]...

    Run database migration commands for a Flask application.

Options:
    --help  Show this message and exit.

Commands:
    create     Create a migration based on an existing model.
    delete     Delete the target migration from the filesystem and database.
    downgrade  Run database downgrades.
    info       Show information about the current database.
    revision   Create a blank migration file.
    status     Show information about the database.
    upgrade    Run database upgrades.
```

This should look very similar since it uses the same commands we just looked at!

For example, to create the migration for User model would look like this:

```
$ flask db create models.User
INFO: created: 0003_create_table_user
```

And to create a blank migration with a custom name would look like this:

```
$ flask db revision "custom revision"
INFO: created: 0004_custom_revision
```

## 5.5 API Documentation

### 5.5.1 Database Manager

**class DatabaseManager**(*database*, *table_name=None*, *directory='migrations'*)

A DatabaseManager is a class responsible for managing and running all migrations against a specific database with a set of migration files.

**Parameters**

- **database** – Connection string, dict, or peewee.Database instance to use.
- **table_name** – Table name to hold migrations (default migration_history).
- **directory** – Directory to store migrations (default migrations).

**create**(*modelstr*)

Create a new migration file for an existing model. Model could actually also be a module, in which case all Peewee models are extracted from the model and created.

**Parameters** **modelstr** – Python class, module, or string pointing to a class or module.

**Returns** True if migration file was created, otherwise False.

**Type** bool

**db_migrations**

List all the migrations applied to the database.

**Returns** List of migration names.

**Return type** tuple

**delete**(*migration*)

Delete the migration from filesystem and database. As if it never happened.

**Parameters** **migration** – Name of migration to find (not including extension).

**Returns** True if delete was successful, otherwise False.

**Return type** bool

**diff**

List all the migrations that have not been applied to the database.

**Returns** List of migration names.

**Return type** tuple

**downgrade**(*target=None*, *fake=False*)

Run all the migrations (down to target if specified). If no target, run one downgrade.

**Parameters**

- **target** – Migration target to limit downgrades.
- **fake** – Should the migration actually run?.

**Returns** True if downgrade was successful, otherwise False.

**Return type** bool

**find_migration**(*value*)
Try to find a migration by name or start of name.

**Raises** ValueError if no matching migration is found.

**Returns** Name of matching migration.

**Return type** str

**get_filename**(*migration*)
Return the full path and filename for the given migration.

**Parameters** **migration** – Name of migration to find (not including extension).

**Returns** Path and filename to migration.

**Return type** str

**get_ident**()
Return a unique identifier for a revision. This defaults to the current next incremental identifier. Override this method to change functionality. Make sure the IDs will be sortable (like timestamps or incremental numbers).

**Returns** Name of new migration.

**Return type** str

**info**()
Show the current database. Don't include any sensitive information like passwords.

**Returns** String representation.

**Return type** str

**load_database**(*database*)
Load the given database, whatever it might be.

A connection string: `sqlite:///database.sqlite`

A dictionary: `{'engine': 'SqliteDatabase', 'name': 'database.sqlite'}`

A peewee.Database instance: `peewee.SqliteDatabase('database.sqlite')`

**Parameters** **database** – Connection string, dict, or peewee.Database instance to use.

**Raises** peewee.DatabaseError if database connection cannot be established.

**Returns** Database connection.

**Return type** peewee.Database instance.

**migration_files**
List all the migrations sitting on the filesystem.

**Returns** List of migration names.

**Return type** tuple

**next_migration**(*name*)
Get the name of the next migration that should be created.

**Parameters** **name** – Name to use for migration (not including identifier).

**Returns** Name of new migration.

**Return type** str

**open_migration**(*migration*, *mode='r'*)
Open a migration file with the given mode and return it.

**Parameters**

- **migration** – Name of migration to find (not including extension).
- **mode** – Mode to pass to open(). Most likely 'r' or 'w'.

**Raises** IOError if the file cannot be opened.

**Returns** File instance.

**Return type** io.FileIO

**revision**(*name=None*)
Create a single blank migration file with given name or default of 'auto migration'.

**Parameters** **name** – Name of migration to create (default auto migration).

**Returns** True if migration file was created, otherwise False.

**Type** bool

**run_migration**(*migration*, *direction='upgrade'*, *fake=False*)
Run a single migration. Does not check to see if migration has already been applied.

**Parameters** **migration** – Migration to run.

**Param** Direction to run (either 'upgrade' or 'downgrade') (default upgrade).

**Returns** True if migration was run successfully, otherwise False.

**Type** bool

**status**()
Show all the migrations and a status for each.

**Returns** True if listing was successful, otherwise None.

**Return type** bool

**upgrade**(*target=None*, *fake=False*)
Run all the migrations (up to target if specified). If no target, run all upgrades.

**Parameters**

- **target** – Migration target to limit upgrades.
- **fake** – Should the migration actually run?.

**Returns** True if upgrade was successful, otherwise False.

**Return type** bool

**write_migration**(*migration*, *name*, *upgrade='pass'*, *downgrade='pass'*)
Open a migration file and write the given attributes to it.

**Parameters** **migration** – Name of migration to find (not including extension).

**Name** Name to write in file header.

**Upgrade** Text for upgrade operations.

**Downgrade** Text for downgrade operations.

**Raises** IOError if the file cannot be opened.

**Returns** None.

**class MigrationHistory**(**args*, ***kwargs*)

Base model to manage migration history in a database. You can use this manually to query the database if you want, but normally it's handled by the DatabaseManager class.

**DoesNotExist**

alias of `MigrationHistoryDoesNotExist`

**date_applied = <DateTimeField: MigrationHistory.date_applied>**

**id = <AutoField: MigrationHistory.id>**

**name = <CharField: MigrationHistory.name>**

### 5.5.2 Migrator

**class Migrator**(*database*)

A migrator is a class that runs migrations for a specific upgrade or downgrade operation.

An instance of this class is automatically created and passed as the only argument to `upgrade(migrator)` and `downgrade(migrator)` methods in migration files.

**Parameters** **database** – Connection string, dict, or peewee.Database instance to use.

**add_column**(*table*, *name*, *coltype*, ***kwargs*)

Add the given column to the given table.

**Parameters**

- **table** – Table name to add column to.
- **name** – Name of the column field to add.
- **coltype** – Column type (from FIELD_TO_PEEWEE).
- **kwargs** – Arguments for the given column type.

**Returns** None

**add_index**(*table*, *columns*, *unique=False*)

Add an index to a table based on columns.

**Parameters**

- **table** – Table name.
- **columns** – Columns (list or tuple).
- **unique** – True or False whether index should be unique (default False).

**Returns** None

**add_not_null**(*table*, *column*)

Add a NOT NULL constraint to a column.

**Parameters**

- **table** – Table name.
- **column** – Column name.

**Returns** None

**create_table**(***kwds*)

Context manager to create the given table. Yield a TableCreator instance on which you can perform operations and add columns.

**Parameters**

- **name** – Name of table to created
- **safe** – If True, will be created with "IF NOT EXISTS" (default False).

**Returns** generator

**Return type** *TableCreator*

**drop_column**(*table*, *name*, *cascade=True*)
Drop the column from the given table.

**Parameters**

- **table** – Table name to drop column from.
- **name** – Name of the column field to drop.
- **cascade** – If True, drop will be cascaded.

**Returns** None

**drop_index**(*table*, *index_name*)
Remove an index from a table.

**Parameters**

- **table** – Table name.
- **index_name** – Index name.

**Returns** None

**drop_not_null**(*table*, *column*)
Remove a NOT NULL constraint to a column.

**Parameters**

- **table** – Table name.
- **column** – Column name.

**Returns** None

**drop_table**(*name*, *safe=False*, *cascade=False*)
Drop the given table.

**Parameters**

- **name** – Table name to drop.
- **safe** – If True, exception will be raised if table does not exist.
- **cascade** – If True, drop will be cascaded.

**Returns** None

**execute_sql**(*sql*, *params=None*)
Run the given sql and return a cursor.

**Parameters**

- **sql** – SQL string.
- **params** – Parameters for the given SQL (default None).

**Returns** SQL cursor

**Return type** cursor

**rename_column**(*table*, *old_name*, *new_name*)
Rename a column leaving everything else in tact.

**Parameters**

- **table** – Table name to rename column from.
- **old_name** – Old column name.
- **new_name** – New column name.

**Returns** None

**rename_table**(*old_name*, *new_name*)
Rename a table leaving everything else in tact.

**Parameters**

- **old_name** – Old table name.
- **new_name** – New table name.

**Returns** None

**class TableCreator**(*name*)
A class used for creating a table in a migration file.

**Parameters** **name** – Name of database table.

**add_constraint**(*value*)
Add a constraint to the model.

**Parameters** **name** – String value of constraint.

**Returns** None

**add_index**(*columns*, *unique=False*)
Add an index to the model.

**Parameters**

- **columns** – Columns (list or tuple).
- **unique** – True or False whether index should be unique (default False).

**bare**(*name*, ***kwargs*)
Create a bare column. Alias for `table.column('bare')`

**biginteger**(*name*, ***kwargs*)
Create a biginteger column. Alias for `table.column('biginteger')`

**bin_uuid**(*name*, ***kwargs*)
Create a binary uuid column. Alias for `table.column('bin_uuid')`

**binary**(*name*, ***kwargs*)
Create a binary column. Alias for `table.column('binary')`

**blob**(*name*, ***kwargs*)
Create a blob column. Alias for `table.column('blob')`

**bool**(*name*, ***kwargs*)
Create a bool column. Alias for `table.column('bool')`

**build_fake_model**(*name*)
Build a fake model with some defaults and the given table name. We need this so we can perform operations that actually require a model class.

**Parameters** **name** – Name of database table.

**Returns** A new model class.

**Return type** peewee.Model

**char**(*name*, ***kwargs*)
Create a char column. Alias for `table.column('char')`

**column**(*coltype*, *name*, ***kwargs*)
Generic method to add a column of any type.

**Parameters**

- **coltype** – Column type (from FIELD_TO_PEEWEE).
- **name** – Name of column.
- **kwargs** – Arguments for the given column type.

**date**(*name*, ***kwargs*)
Create a date column. Alias for `table.column('date')`

**datetime**(*name*, ***kwargs*)
Create a datetime column. Alias for `table.column('datetime')`

**decimal**(*name*, ***kwargs*)
Create a decimal column. Alias for `table.column('decimal')`

**double**(*name*, ***kwargs*)
Create a double column. Alias for `table.column('double')`

**fixed**(*name*, ***kwargs*)
Create a fixed column. Alias for `table.column('fixed')`

**float**(*name*, ***kwargs*)
Create a float column. Alias for `table.column('float')`

**foreign_key**(*coltype*, *name*, *references*, ***kwargs*)
Add a foreign key to the model. This has some special cases, which is why it's not handled like all the other column types.

**Parameters**

- **name** – Name of the foreign key.
- **references** – Table name in the format of "table.column" or just "table" (and id will be default column).
- **kwargs** – Additional kwargs to pass to the column instance. You can also provide "on_delete" and "on_update" to add constraints.

**Returns** None

**int**(*name*, ***kwargs*)
Create a int column. Alias for `table.column('int')`

**integer**(*name*, ***kwargs*)
Create a integer column. Alias for `table.column('integer')`

**primary_key**(*name*)
Add a primary key to the model. This has some special cases, which is why it's not handled like all the other column types.

**Parameters** **name** – Name of column.

**Returns** None

**smallint**(*name*, ***kwargs*)
Create a smallint column. Alias for `table.column('smallint')`

**smallinteger**(*name*, ***kwargs*)
Create a smallinteger column. Alias for `table.column('smallinteger')`

**text**(*name*, ***kwargs*)
Create a text column. Alias for `table.column('text')`

**time**(*name*, ***kwargs*)
Create a time column. Alias for `table.column('time')`

**uuid**(*name*, ***kwargs*)
Create a uuid column. Alias for `table.column('uuid')`

### 5.5.3 CLI Functions

**cli_command**(**args*, ***kwargs*)
Run database migration commands.

**cli_info**(**args*, ***kwargs*)
Show information about the current database.

**cli_status**(**args*, ***kwargs*)
Show information about migration status.

**cli_create**(**args*, ***kwargs*)
Create a migration based on an existing model.

**cli_revision**(**args*, ***kwargs*)
Create a blank migration file.

**cli_upgrade**(**args*, ***kwargs*)
Run database upgrades.

**cli_downgrade**(**args*, ***kwargs*)
Run database downgrades.

**cli_delete**(**args*, ***kwargs*)
Delete the target migration from the filesystem and database.

## 5.6 Changelog

### 5.6.1 2.1.0

- Added *–fake* flag for upgrade and downgrade operations.

### 5.6.2 2.0.3

- Handle null=True on ForeignKeyField.

### 5.6.3 2.0.2

- Fix for indexes created with "fakemodel" prefix.

### 5.6.4 2.0.1

- Fix for TimestampField.

### 5.6.5 2.0.0

- Added support for Peewee 3.
- Dropped support for Peewee 2.

### 5.6.6 1.7.3

- Added missing click dependency.

### 5.6.7 1.7.2

- Fixed issue passing a `peewee.Proxy` to `DatabaseManager`.

### 5.6.8 1.7.1

- Added `get_flask_database_manager` helper.

### 5.6.9 1.7.0

- Major rewrite and first release with proper documentation.

CHAPTER 6

# License

# Index

## O



## P



## R



## S



## T



## U



## W