

---

# Django Evolution Documentation

**Beanbag, Inc.**

**Oct 15, 2023**



# CONTENTS

<b>1 Frequently Asked Questions</b>	<b>3</b>
1.1 Who maintains Django Evolution? . . . . .	3
1.2 Where do I go for support? . . . . .	3
1.3 What about bug reports? . . . . .	3
1.4 How do I contribute patches/pull requests? . . . . .	3
1.5 Why evolutions and not migrations? . . . . .	4
1.6 Can I switch apps from evolutions to migrations? . . . . .	4
1.7 Can I switch apps from migrations to evolutions? . . . . .	4
1.8 Why do my syncdb/migrate commands act differently? . . . . .	5
<b>2 Installing Django Evolution</b>	<b>7</b>
<b>3 Writing Evolutions</b>	<b>9</b>
3.1 Example . . . . .	9
<b>4 App and Model Mutations</b>	<b>15</b>
4.1 Field Mutations . . . . .	15
4.2 Model Mutators . . . . .	18
4.3 App Mutators . . . . .	20
4.4 Other Mutators . . . . .	21
<b>5 Management Commands</b>	<b>23</b>
5.1 evolution-project-sig . . . . .	23
5.2 evolve . . . . .	24
5.3 list-evolutions . . . . .	25
5.4 mark-evolution-applied . . . . .	26
5.5 wipe-evolution . . . . .	26
<b>6 Glossary</b>	<b>29</b>
<b>7 Release Notes</b>	<b>31</b>
7.1 2.x Releases . . . . .	31
7.2 0.7 Releases . . . . .	39
7.3 0.6 Releases . . . . .	46
7.4 0.5 Releases . . . . .	51
<b>8 Django Evolution Documentation</b>	<b>53</b>
8.1 Questions So Far? . . . . .	53
8.2 Let's Get Started . . . . .	54
8.3 Reference . . . . .	54

**Python Module Index** **223**

**Index** **225**

Django Evolution tracks all the apps in your Django project, recording information on the structure of models, their fields, indexes, and so on.

When you make any change to a model that would need to be reflected in the database, Django Evolution will tell you that you'll need an evolution file to apply those changes, and will suggest one for you.

Evolution files describe one or more changes made to models in an app. They can:

- Add fields
- Change the attributes on fields
- Rename fields
- Delete fields
- Change the indexes or constraints on a model
- Rename models
- Delete models
- Rename apps
- Delete apps
- Transition an app to Django's migrations
- Run arbitrary SQL

Django Evolution looks at the last-recorded state of your apps, the current state, and the evolution files. If those evolution files are enough to update the database to the current state, then Django Evolution will process them, turning them into an optimized list of SQL statements, and apply them to the database.

This can be done for the entire database as a whole, or for specific apps in the database.

Since some apps (particularly Django's own apps) make use of migrations (on Django 1.7 and higher), Django Evolution will also handle applying those migrations. It will do this in cooperation with the evolution files that it also needs to apply. However, it's worth pointing out that migrations are never optimized the way evolutions are (this is currently a limitation in Django).



## FREQUENTLY ASKED QUESTIONS

### 1.1 Who maintains Django Evolution?

Originally, Django Evolution was built by two guys in Perth, Australia: [Ben Khoo](#) and Russell Keith-Magee (a core developer on Django).

Since then, Django Evolution has been taken over by [Beanbag, Inc.](#). We have a vested interest in keeping this alive, well-maintained, and open source for [Review Board](#) and other products.

### 1.2 Where do I go for support?

We have a really old [mailing list](#) over at Google Groups, where you can ask questions. Truthfully, this group is basically empty these days, but you can still ask there and we'll answer!

We also provide commercial support. You can [reach out to us](#) if you're using Django Evolution in production and want the assurance of someone you can reach 24/7 if something goes wrong.

### 1.3 What about bug reports?

You can report bugs on our [bug tracker](#), hosted on [Splat](#).

When you file a bug, please be as thorough as possible. Ideally, we'd like to see the contents of your `django_project_version` and `django_evolution` tables before and after the upgrade, along with any evolution files, models, and error logs.

### 1.4 How do I contribute patches/pull requests?

We'd love to work with you on your contributions to Django Evolution! It'll make our lives easier, for sure :)

While we don't work with pull requests, we do accept patches on [reviews.reviewboard.org](#), our [Review Board](#) server. You can get started by cloning our [GitHub repository](#), and [install RBTools](#) (the Review Board command line tools).

To post new changes from your feature branch for review, run:

```
$ rbt post
```

To update an existing review request:

```
$ rbt post -u
```

See the [RBTools documentation](#) for more usage info.

## 1.5 Why evolutions and not migrations?

While most new projects would opt for Django's own *migrations*, there are a few advantages to using evolutions:

1. Evolutions are faster to apply than migrations when upgrading between arbitrary versions of the schema.

Migrations are applied one at a time. If you have 10 migrations modifying one table, then you'll trigger a table rebuild 10 times, which is slow – particularly if there's a lot of data in that table.

Evolutions going through an optimization process before they're applied, determining the smallest amount of changes needed. 10 evolutions for a table will generally only trigger a single table rebuild.

When you fully own the databases you're upgrading, this may not matter, as you're probably applying new migrations as you write them. However, if you are distributing self-installed web services (such as [Review Board](#)), administrators may not upgrade often. Evolutions help keep these large upgrades from taking forever.

2. There's a wide range of Django support.

If you are still maintaining legacy applications on Django 1.6, it may be hard to transition to newer versions. By switching to Django Evolution, there's a transition path. You can use evolutions for the apps you control without conflicting with migrations, and begin the upgrade path to modern versions of Django.

At any time, you can easily switch some or all of your apps from evolutions to migrations, and Django Evolution will take care of it automatically.

3. Django Evolution is easier for some development processes.

During development, you may make numerous changes to your database, necessitating schema changes that you wouldn't want to apply in production. With migrations, you'd need to squash those development-only migration files, which doesn't play as well if some beta users have only a subset of those migrations applied.

## 1.6 Can I switch apps from evolutions to migrations?

Yes, you can! The [\*MoveToDjangoMigrations\*](#) mutation will instruct Django Evolution to use *migrations* instead of evolutions for any future changes. Before it hands your app off entirely, it will apply any unapplied evolutions, ensuring a sane starting point for your new migrations.

## 1.7 Can I switch apps from migrations to evolutions?

No, it's one way for now. We might add this if anyone wants it in the future. For now, we assume that people using migrations are satisfied with that, and aren't looking to move to evolutions.

## 1.8 Why do my syncdb/migrate commands act differently?

Starting in Django Evolution 2.0, the `evolve` command has taken over all responsibilities for creating and updating the database, replacing `syncdb` and `migrate`.

For compatibility, those two commands have been replaced, wrapping `evolve` instead. Some functionality had to be stripped away from the original commands, though.

Our `syncdb` and `migrate` commands don't support loading `initial_data` fixtures. This feature was deprecated in Django 1.7 and removed in 1.9, and keeping support between Django versions is tricky. We've opted not to include it (at least for now).

Our `migrate` command doesn't support specifying explicit migration names to apply, or using `--fake` to pretend migrations were applied.

It's possible we'll add compatibility in the future, but only if demand is strong.



---

CHAPTER  
**TWO**

---

## **INSTALLING DJANGO EVOLUTION**

To install Django Evolution, simply run:

```
$ pip install django_evolution
```

You'll probably want to add that as a package dependency to your project.

Then add `django_evolution` to your project's `INSTALLED_APPS`.



## WRITING EVOLUTIONS

Evolution files describe a set of changes made to an app or its models. These are Python files that live in the `appdir/evolutions/` directory. The name of the file (minus the `.py` extension) is called an *evolution label*, and can be whatever you want, so long as it's unique for the app. These files look something like:

Listing 1: myapp/evolutions/my\_evolution.py

```
from __future__ import unicode_literals

from django_evolution.mutations import AddField

MUTATIONS = [
    AddField('MyModel', 'my_field', models.CharField, max_length=100,
            null=True),
]
```

Evolution files can make use of any supported *App and Model Mutations* (classes like `AddField` above) to describe the changes made to your app or models.

Once you've written an evolution file, you'll need to place its label in the app's `appdir/evolutions/__init__.py` in a list called `SEQUENCE`. This specifies the order in which evolutions should be processed. These look something like:

Listing 2: myapp/evolutions/\_\_init\_\_.py

```
from __future__ import unicode_literals

SEQUENCE = [
    'my_evolution',
]
```

### 3.1 Example

Let's go through an example, starting with a model.

Listing 3: blogs/models.py

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
    date_of_birth = models.DateField()
```

(continues on next page)

(continued from previous page)

```
class Entry(models.Model):
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateTimeField()
    author = models.ForeignKey(Author)
```

At this point, we'll assume that the project has been previously synced to the database using something like `./manage.py syncdb` or `./manage.py migrate --run-syncdb`. We will also assume that it does *not* make use of `migrations`.

### 3.1.1 Modifying Our Model

Perhaps we decide we don't actually need the birthdate of the author. It's just extra data we're doing nothing with, and increases the maintenance burden. Let's get rid of it.

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()
-   date_of_birth = models.DateField()
```

The field is gone, but it's still in the database. We need to generate an evolution to get rid of it.

We can get a good idea of what this should look like by running:

```
$ ./manage.py evolve --hint
```

Which gives us:

```
----- Evolution for blogs
from __future__ import unicode_literals

from django_evolution.mutations import DeleteField

MUTATIONS = [
    DeleteField('Author', 'date_of_birth'),
]
-----

Trial upgrade successful!
```

As you can see, we got some output showing us what the evolution file might look like to delete this field. We're also told that this worked – this evolution was enough to update the database based on our changes. If we had something more complex (like adding a non-null field, requiring some sort of initial value), then we'd be told we still have changes to make.

Let's dump this sample file in `blogs/evolutions/remove_date_of_birth.py`:

Listing 4: blogs/evolutions/remove\_date\_of\_birth.py

```
from __future__ import unicode_literals
```

(continues on next page)

(continued from previous page)

```
from django_evolution.mutations import DeleteField

MUTATIONS = [
    DeleteField('Author', 'date_of_birth'),
]
```

(Alternatively, we could have run `./manage.py evolve -w remove_date_of_birth`, which would create this file for us, but let's start off this way.)

Now we need to tell Django Evolution we want this in our evolution sequence:

Listing 5: blogs/evolutions/remove\_date\_of\_birth.py

```
from __future__ import unicode_literals

SEQUENCE = [
    'remove_date_of_birth',
]
```

We're done with the hard work! Time to apply the evolution:

```
$ ./manage.py evolve --execute

You have requested a database upgrade. This will alter tables and data
currently in the "default" database, and may result in IRREVERSABLE
DATA LOSS. Upgrades should be *thoroughly* reviewed and tested prior
to execution.

MAKE A BACKUP OF YOUR DATABASE BEFORE YOU CONTINUE!

Are you sure you want to execute the database upgrade?

Type "yes" to continue, or "no" to cancel: yes

This may take a while. Please be patient, and DO NOT cancel the
upgrade!

Applying database evolution for blogs...
The database upgrade was successful!
```

Tada! Now if you look at the columns for your `blogs_author` table, you'll find that `date_of_birth` is gone.

You can make changes to your models as often as you need to. Add and delete the same field a dozen times across dozens of evolutions, if you like. Evolutions are automatically optimized before applied, resulting in the smallest set of changes needed to get your database updated.

### 3.1.2 Adding Dependencies

New in version 2.1.

Both individual evolution modules and the main `myapp/evolutions/__init__.py` module can define other evolutions or migrations that must be applied before or after the individual evolution or app as a whole.

This is done by adding any of the following to the appropriate module:

#### AFTER\_EVOLUTIONS:

A list of specific evolutions (tuples in the form of `(app_label, evolution_label)`) or app labels (a single string) that must be applied before this evolution can be applied.

#### BEFORE\_EVOLUTIONS:

A list of specific evolutions (tuples in the form of `(app_label, evolution_label)`) or app labels (a single string) that must be applied sometime after this evolution is applied.

#### AFTER\_MIGRATIONS:

A list of migration targets (tuples in the form of `(app_label, migration_name)`) that must be applied before this evolution can be applied.

#### BEFORE\_MIGRATIONS:

A list of migration targets (tuples in the form of `(app_label, migration_name)`) that must be applied sometime after this evolution is applied.

Django Evolution will apply the evolutions and migrations in the right order based on any dependencies.

This is important to set if you have evolutions that a migration may depend on (e.g., a swappable model that the migration requires), or if your evolutions are being applied in the wrong order (often only a problem if there are evolutions depending on migrations).

---

**Note:** It's up to you to decide where to put these.

You may want to define this as its own empty `initial.py` evolution at the beginning of the SEQUENCE list, or to a more specific evolution within.

---

So, let's look at an example:

Listing 6: blogs/evolutions/add\_my\_field.py

```
from __future__ import unicode_literals

from django_evolution.mutations import ...

BEFORE_EVOLUTIONS = [
    'blog_exporter',
    ('myapi', 'add_blog_fields'),
]

AFTER_MIGRATIONS = [
    ('fancy_text', '0001_initial'),
]

MUTATIONS = [
    ...
]
```

This will ensure this evolution is applied before both the `blog_exporter` app's evolutions/models and the `myapi` app's `add_blog_fields` evolution. At the same time, it'll also ensure that it will be applied only after the `fancy_text` app's `0001_initial` migration has been applied.

Similarly, these can be added to the top-level `evolutions/__init__.py` file for an app:

Listing 7: blogs/evolutions/\_\_init\_\_.py

```
from __future__ import unicode_literals

BEFORE_EVOLUTIONS = [
    'blog_exporter',
    ('myapi', 'add_blog_fields'),
]

AFTER_MIGRATIONS = [
    ('fancy_text', '0001_initial'),
]

SEQUENCE = [
    'add_my_field',
]
```

This is handy if you need to be sure that this module's evolutions or model creations always happen before or after that of another module, no matter which models may exist or which evolutions may have already been applied.

---

**Hint:** Don't add dependencies if you don't need to. Django Evolution will try to apply the ordering in the correct way. Use dependencies when it gets it wrong.

Make sure you test not only upgrades but the creation of brand-new databases, to make sure your dependencies are correct in both cases.

---

## MoveToDjangoMigrations

If an evolution uses the `MoveToDjangoMigrations` mutation, dependencies will automatically be created to ensure that your evolution is applied in the correct order relative to any new migrations in that app.

That means that this:

```
MUTATIONS = [
    MoveToDjangoMigrations(mark_applied=['0001_initial'])
]
```

implies:

```
AFTER_MIGRATIONS = [
    ('myapp', '0001_initial'),
]
```



## APP AND MODEL MUTATIONS

Evolutions are composed of one or more mutations, which mutate the state of the app or models. There are several mutations included with Django Evolution, which we'll take a look at here.

### 4.1 Field Mutations

#### 4.1.1 AddField

AddField is used to add new fields to a table. It takes the following parameters:

```
class AddField(model_name, field_name, field_type, initial=None, **fieldAttrs)
```

##### Parameters

- **model\_name** (*str*) – The name of the model the field was added to.
- **field\_name** (*str*) – The name of the new field.
- **field\_type** (*type*) – The field class.
- **initial** – The initial value to set for the field. Each row in the table will have this value set once the field is added. It's required if the field is non-null.
- **fieldAttrs** (*dict*) – Attributes to pass to the field constructor. Only those that impact the schema of the table are considered (for instance, `null=True` or `max_length=100`, but not `help_text="..."`).

For example:

```
from django.db import models
from django_evolution.mutations import AddField

MUTATIONS = [
    AddField('Book', 'publish_date', models.DateTimeField, null=True),
]
```

## 4.1.2 ChangeField

ChangeField can make changes to existing fields, altering the attributes (for instance, increasing the maximum length of a CharField).

---

**Note:** This cannot be used to change the field type.

---

It takes the following parameters:

```
class ChangeField(model_name, field_name, initial=None, **field_attrs)
```

### Parameters

- **model\_name** (*str*) – The name of the model containing the field.
- **field\_name** (*str*) – The name of the field to change.
- **field\_type** – The new type of field. This must be a subclass of [Field](#).

This will do its best to change one field type to another, but not all field types can be changed to another type. Some types may be database-specific.

New in version 2.2.

- **initial** – The new initial value to set for the field. If the field previously allowed null values, but null=False is being passed, then this will update all existing rows in the table to have this initial value.
- **fieldAttrs** (*dict*) – The field attributes to change. Only those that impact the schema of the table are considered (for instance, null=... or max\_length=..., but not help\_text=....)

For example:

```
from django.db import models
from django_evolution.mutations import ChangeField

MUTATIONS = [
    ChangeField('Book', 'name', max_length=100, null=False),
]
```

## 4.1.3 DeleteField

DeleteField will delete a field from the table, erasing its data from all rows. It takes the following parameters:

```
class DeleteField(model_name, field_name)
```

### Parameters

- **model\_name** (*str*) – The name of the model containing the field to delete.
- **field\_name** (*str*) – The name of the field to delete.

For example:

```
from django.db import models
from django_evolution.mutations import ChangeField

MUTATIONS = [
    ChangeField('Book', 'name', max_length=100, null=False),
]
```

#### 4.1.4 RenameField

`RenameField` will rename a field in the table, preserving all stored data. It can also set an explicit column name (in case the name is only changing in the model) or a `ManyToManyField` table name.

If working with a `ManyToManyField`, then the parent table won't actually have a real column backing it. Instead, the relationships are all maintained using the "through" table created by the field. In this case, the `db_column` value will be ignored, but `db_table` can be set.

It takes the following parameters:

```
class RenameField(model_name, old_field_name, new_field_name, db_column=None, db_table=None)
```

##### Parameters

- `model_name` (`str`) – The name of the model containing the field to delete.
- `old_field_name` (`str`) – The old name of the field on the model.
- `new_field_name` (`str`) – The new name of the field on the model.
- `db_column` (`str`) – The explicit name of the column on the table to use. This may be the original column name, if the name is only being changed on the model (which means no database changes may be made).
- `db_table` (`str`) – The explicit name of the "through" table to use for a `ManyToManyField`. If changed, then that table will be renamed. This is ignored for any other types of fields.

If the table name hasn't actually changed, then this may not make any changes to the database.

For example:

```
from django_evolution.mutations import RenameField

MUTATIONS = [
    RenameField('Book', 'isbn_number', 'isbn', column_name='isbn_number'),
    RenameField('Book', 'critics', 'reviewers',
               db_table='book_critics')
]
```

## 4.2 Model Mutators

### 4.2.1 ChangeMeta

ChangeMeta can change certain bits of metadata about a model. For example, the indexes or unique-together constraints. It takes the following parameters:

```
class ChangeMeta(model_name, prop_name, new_value)
```

#### Parameters

- **model\_name** (*str*) – The name of the model containing the field to delete.
- **prop\_name** (*str*) – The name of the property to change, as documented below.
- **new\_value** – The new value for the property.

The properties that can be changed depend on the version of Django. They include:

#### db\_table\_comment:

A comment to apply to the table's schema.

This requires Django 4.2 or higher.

#### Version Added:

2.3

#### index\_together:

Groups of fields that should be indexed together in the database.

This is represented by a list of tuples, each of which groups together multiple field names that should be indexed together in the database.

index\_together support requires Django 1.5 or higher. The last versions of Django Evolution to support Django 1.5 was the 0.7.x series.

#### indexes:

Explicit indexes to create for the model, optionally grouping multiple fields together and optionally naming the index.

This is represented by a list of dictionaries, each of which contain a `fields` key and an optional `name` key. Both of these correspond to the matching fields in Django's `Index` class.

indexes support requires Django 1.11 or higher.

#### unique\_together:

Groups of fields that together form a unique constraint. Rows in the database cannot repeat the same values for those groups of fields.

This is represented by a list of tuples, each of which groups together multiple field names that should be unique together in the database.

unique\_together support is available in all supported versions of Django.

For example:

```
from django_evolution.mutations import ChangeMeta

MUTATIONS = [
    ChangeMeta('Book', 'index_together', [('name', 'author')]),
]
```

Changed in version 2.0: Added support for indexes.

## 4.2.2 DeleteModel

DeleteModel removes a model from the database. It will also remove any “through” models for any of its `ManyToManyFields`. It takes the following parameters:

```
class DeleteModel(model_name)
```

### Parameters

`model_name (str)` – The name of the model to delete.

For example:

```
from django_evolution.mutations import DeleteModel

MUTATIONS = [
    DeleteModel('Book'),
]
```

## 4.2.3 RenameModel

RenameModel will rename a model and update all relations pointing to that model. It requires an explicit underlying table name, which can be set to the original table name if only the Python-side model name is changing. It takes the following parameters:

```
class RenameModel(old_model_name, new_model_name, db_table)
```

### Parameters

- `old_model_name (str)` – The old name of the model.
- `new_model_name (str)` – The new name of the model.
- `db_table (str)` – The explicit name of the underlying table.

For example:

```
from django_evolution.mutations import RenameModel

MUTATIONS = [
    RenameModel('Critic', 'Reviewer', db_table='books_reviewer'),
]
```

## 4.3 App Mutators

### 4.3.1 DeleteApplication

`DeleteApplication` will remove all the models for an app from the database, erasing all associated data. This mutation takes no parameters.

---

**Note:** Make sure that any relation fields from other models to this app's models have been removed before deleting an app.

In many cases, you may just want to remove the app from your project's `INSTALLED_APPS`, and leave the data alone.

---

For example:

```
from django_evolution.mutations import DeleteApplication

MUTATIONS = [
    DeleteApplication(),
]
```

### 4.3.2 MoveToDjangoMigrations

`MoveToDjangoMigrations` will tell Django Evolution that any future changes to the app or its models should be handled by Django's `migrations` instead evolutions. Any unapplied evolutions will be applied before applying any migrations.

This is a one-way operation. Once an app moves from evolutions to migrations, it cannot move back.

Since an app may have had both evolutions and migrations defined in the tree (in order to work with both systems), this takes a `mark_applied=` parameter that lists the migrations that should be considered applied by the time this mutation is run. Those migrations will be recorded as applied and skipped.

```
class MoveToDjangoMigrations(mark_applied=['0001_initial'])
```

#### Parameters

`mark_applied` (`list`) – The list of migrations that should be considered applied when running this mutation. This defaults to the `0001_initial` migration.

For example:

```
from django_evolution.mutations import MoveToDjangoMigrations

MUTATIONS = [
    MoveToDjangoMigrations(mark_applied=['0001_initial',
                                         '0002_book_add_isbn']),
]
```

New in version 2.0.

### 4.3.3 RenameAppLabel

`RenameAppLabel` will rename the stored app label for the app, updating all references made in other models. It won't change indexes or any database state, however.

Django 1.7 moved to an improved concept of app labels that could be customized and were guaranteed to be unique within a project (we'll call these *modern app labels*). Django 1.6 and earlier generated app labels based on the app's module name (*legacy app labels*).

Because of this, older stored *project signatures* may have grouped together models from two different apps (both with the same app labels) together. Django Evolution will *try* to untangle this, but in complicated cases, you may need to supply a list of model names for the app (current and possibly older ones that have been removed). Whether you need to do this is entirely dependent on the structure of your project. Test it in your upgrades.

This takes the following parameters:

```
class RenameAppLabel(old_app_label, new_app_label, legacy_app_label=None, model_names=None)
```

#### Parameters

- **old\_app\_label** (*str*) – The old app label that's being renamed.
- **new\_app\_label** (*str*) – The new modern app label to rename to.
- **legacy\_app\_label** (*str*) – The legacy app label for the new app name. This provides compatibility with older versions of Django and helps with transition apps and models.
- **model\_names** (*list*) – The list of model names to move out of the old signature and into the new one.

For example:

```
from django_evolution.mutations import RenameAppLabel

MUTATIONS = [
    RenameAppLabel('admin', 'my_admin', legacy_app_label='admin',
                  model_names=['Report', 'Config']),
]
```

New in version 2.0.

## 4.4 Other Mutators

### 4.4.1 SQLMutation

`SQLMutation` is an advanced mutation used to make arbitrary changes to a database and to the stored project signature. It may be used to make changes that cannot be made by other mutators, such as altering tables not managed by Django, changing a table engine, making metadata changes to the table or database, or modifying the content of rows.

SQL from this mutation cannot be optimized alongside other mutations.

This takes the following parameters:

```
class SQLMutation(tag, sql, update_func=None)
```

#### Parameters

- **tag** (*str*) – A unique identifier for this SQL mutation within the app.

- **sql**(*list/str*) – A list of SQL statements, or a single SQL statement as a string, to execute. Note that this will be database-dependent.
- **update\_func**(*callable*) – A function to call to perform additional operations or update the *project signature*.

---

**Note:** There's some caveats with providing an `update_func`.

Django Evolution 2.0 introduced a new form for this function that takes in a `django_evolution.mutations.Simulation` object, which can be used to access and modify the stored *project signature*. This is safe to use (well, relatively – try not to blow anything up).

Prior versions supported a function that took two arguments: The app label of the app that's being evolved, and a serialized dictionary representing the project signature.

If using the legacy style, it's *possible* that you can mess up the signature data, since we have to serialize to an older version of the signature and then load from that. Older versions of the signature don't support all the data that newer versions do, so how well this works is really determined by the types of evolutions that are going to be run.

We **strongly** recommend updating *any* `SQLMutation` calls to use the new-style function format, for safety and future compatibility.

---

For example:

```
from django_evolution.mutations import SQLMutation

def _update_signature(simulation):
    pass

MUTATIONS = [
    SQLMutation('set_innodb_engine',
                'ALTER TABLE my_table ENGINE = MYISAM;',
                update_func=_update_signature),
]
```

Changed in version 2.0: Added the new-style `update_func`.

## MANAGEMENT COMMANDS

### 5.1 evolution-project-sig

The **evolution-project-sig** command is used to list, show, and delete stored project signatures.

This is really only useful if you're working to recover from a bad state where you've undone the changes made by an evolution and need to re-apply it. It should never be used under normal use, especially on a production database.

By default, this command will confirm before making any changes to the database. You can use `--noinput` to avoid the confirmation step.

#### 5.1.1 Example

To list project signatures:

```
$ ./manage.py evolution-project-sig --list
```

To show the latest project signature:

```
$ ./manage.py evolution-project-sig --show
```

To show a specific project signature:

```
$ ./manage.py evolution-project-sig --show --id <ID>
```

To delete a project signature:

```
$ ./manage.py evolution-project-sig --delete --id <ID>
```

#### 5.1.2 Arguments

##### **--delete**

Delete a project signature.

##### **--list**

List project signatures and their associated evolutions.

##### **--show**

Show the current project signature, or an older one if using `--id`.

**--id**

Specify the ID of a project signature.

**--noinput**

Delete without prompting for confirmation.

## 5.2 evolve

The **evolve** command is responsible for setting up databases and applying any evolutions or *migrations*.

This is a replacement for both the **syncdb** and **migrate** commands in Django. Running either of this will wrap **evolve** (though not all of the command's arguments will be supported when Django Evolution is enabled).

### 5.2.1 Creating/Updating Databases

To construct a new database or apply updates, you will generally just run:

```
$ ./manage.py evolve --execute
```

This is the most common usage for **evolve**. It will create any missing models and apply any unapplied evolutions or *migrations*.

Changed in version 2.0: **evolve** now replaces both **syncdb** and **migrate**. In previous versions, it had to be run after **syncdb**.

### 5.2.2 Generating Hinted Evolutions

When making changes to a model, it helps to see how the evolution should look before writing it. Sometimes the evolution will be usable as-is, but sometimes you'll need to tweak it first.

To generate a hinted evolution, run:

```
$ ./manage.py evolve --hint
```

Hinted evolutions can be automatically written by using **--write**, saving you a little bit of work:

```
$ ./manage.py evolve --hint --write my_new_evolution
```

This will take any app with a hinted evolution and write a *appdir/evolutions/my\_new\_evolution.py* file. You will still need to add your new evolution to the SEQUENCE list in *appdir/evolutions/\_\_init\_\_.py*.

If you only want to write hints for a specific app, pass the app labels on the command line, like so:

```
$ ./manage.py evolve --hint --write my_new_evolution my_app
```

### 5.2.3 Arguments

**<APP\_LABEL...>**

Zero or more specific app labels to evolve. If provided, only these apps will have evolutions or *migrations* applied. If not provided, all apps will be considered for evolution.

**--database <DATABASE>**

The name of the configured database to perform the evolution against.

**--hint**

Display sample evolutions that fulfill any database changes for apps and models managed by evolutions. This won't include any apps or models managed by *migrations*.

**--noinput**

Perform evolutions automatically without any input.

**--purge**

Remove information on any non-existent applications from the stored project signature. This won't remove the models themselves. For that, see [DeleteModel](#) or [DeleteApplication](#).

**--sql**

Display the generated SQL that would be run if applying evolutions. This won't include any apps or models managed by *migrations*.

**-w <EVOLUTION\_NAME>, --write <EVOLUTION\_NAME>**

Write any hinted evolutions to a file named *appdir/evolutions/EVOLUTION\_NAME*. This will *not* include the evolution in *appdir/evolutions/\_\_init\_\_.py*.

**-x, --execute**

Execute the evolution process, applying any evolutions and *migrations* to the database.

**Warning:** This can be used in combination with **--hint** to apply hinted evolutions, but this is generally a **bad idea**, as the execution is not properly repeatable or trackable.

## 5.3 list-evolutions

The **list-evolutions** command lists all the evolutions that have so far been applied to the database. It can be useful for debugging, or determining if a specific evolution has yet been applied.

### 5.3.1 Example

```
$ ./manage.py list-evolutions my_app
Applied evolutions for 'my_app':
    add_special_fields
    update_app_label
    change_name_max_length
```

### 5.3.2 Arguments

**<APP\_LABEL...>**

Zero or more specific app labels to list. If provided, only evolutions on these apps will be shown.

## 5.4 mark-evolution-applied

The **mark-evolution-applied** command is used to mark evolutions as already applied in the database.

This is really only useful if you're working to recover from a bad state where you've undone the changes made by an evolution and need to re-apply it. It should never be used under normal use, especially on a production database.

By default, this command will confirm before marking the evolution as applied. You can use **--noinput** to avoid the confirmation step.

### 5.4.1 Example

```
$ ./manage.py mark-evolution-applied --app-label my_app \
    change_name_max_length
```

### 5.4.2 Arguments

**EVOLUTION\_LABEL ...**

One or more specific evolution labels to mark as applied. This is required if **--all** isn't specified.

**--all**

Mark all unapplied evolutions as applied.

**--app-label <APP\_LABEL>**

An app label the evolutions apply to.

**--noinput**

Mark as applied without prompting for confirmation.

## 5.5 wipe-evolution

The **wipe-evolution** command is used to remove evolutions from the list of applied evolutions.

This is really only useful if you're working to recover from a bad state where you've undone the changes made by an evolution and need to re-apply it. It should never be used under normal use, especially on a production database.

By default, this command will confirm before wiping the evolution from the history. You can use **--noinput** to avoid the confirmation step.

To see the list of evolutions that can be wiped, run **list-evolutions**.

### 5.5.1 Example

```
$ ./manage.py wipe-evolution --app-label my_app change_name_max_length
```

### 5.5.2 Arguments

#### **EVOLUTION\_LABEL ...**

One or more specific evolution labels to remove from the database. If the same evolution names exist for multiple apps, they'll all be removed. To isolate them to a specific app, use [--app-label](#).

#### **--app-label <APP\_LABEL>**

An app label to limit evolution labels to. Only evolutions on this app will be wiped.

#### **--noinput**

Perform the wiping procedure automatically without any input.



---

**CHAPTER  
SIX**

---

## **GLOSSARY**

**evolution label**

The name of a particular evolution for an app. These must be unique within an app, but do not have to be unique within a project.

**legacy app label**

**legacy app labels**

The form of app label used in Django 1.6 and earlier. Legacy app labels are generated solely from the app's module name.

**migrations**

Django 1.7+'s built-in method of managing changes to the database schema. See the [migrations documentation](#).

**modern app label**

**modern app labels**

The form of app label used in Django 1.7 and later. Modern app labels default to being generated from the app's module name, but can be customized.

**project signature**

**project signatures**

A stored representation of all the apps and models in your project. This is stored in the `django_project_version` table, and is a critical part in determining how the database has evolved and what changes need to be made.

In Django Evolution 2.0 and higher, this is stored as JSON data. In prior versions, this was stored as Pickle protocol 0 data.



## RELEASE NOTES

### 7.1 2.x Releases

#### 7.1.1 Django Evolution 2.3

**Release date:** October 15, 2023

##### Installation

Django Evolution 2.3 is compatible with [Django 1.6-4.2](#), and Python 2.7 and 3.6-3.12.

To install Django Evolution 2.3, run:

```
$ pip3 install django_evolution==2.3
```

To learn more, see:

- Documentation
- [Django Evolution on PyPI](#)
- [Django Evolution on GitHub](#)

##### New Features

- Added support for Python 3.12 and Django 4.2.
- Added support for evolving table comments on Django 4.2.

This is done through [\*ChangeMeta\*](#).

- Added advanced management commands for working with project signatures and marking evolutions as applied. [\*mark-evolution-applied\*](#) will mark one or more evolutions as applied to your database, without modifying any schema.

[\*evolution-project-sig\*](#) will let you list project signatures, show a stored project signature, or delete project signatures.

These are advanced and dangerous commands. They should only be run if you know what you're doing, as part of diagnosing and fixing a failed database upgrade.

- Added debug logging for the evolution process.

If Python's logging is set up to enable debug output, then the evolution process will provide information on the new models generation, mutations, and evolutions begin run. This can aid in debugging efforts.

### Contributors

- Christian Hammond
- David Trowbridge

## 7.1.2 Django Evolution 2.2

**Release date:** October 3, 2022

### New Features

- Added support for Django 3.2 through 4.1.

This includes full support for `django.db.models.Index`, and compatibility with database backend changes made in these versions.

- Added support for changing a field's type in `ChangeField`.

This can be done by passing in the new field class to `field_type=...`

- Added a new `settings.DJANGO_EVOLUTION` setting.

This is in the form of:

```
DJANGO_EVOLUTION = {  
    'CUSTOM_EVOLUTIONS': {  
        '<app_label>': ['<evolution_module>', ...],  
    },  
    'ENABLED': <bool>,  
}
```

This replaces `settings.CUSTOM_EVOLUTIONS` and `settings.DJANGO_EVOLUTION_ENABLED`, both of which are now deprecated and will emit deprecation warnings.

### Bug Fixes

#### General

- Fixed generating SQL to execute while in a transaction on Django 2.0+.

#### Indexes/Constraints

- Fixed ordering issues when dropping and re-creating indexes when changing `db_index` and `unique` states.
- Fixed deferring constraints and indexes when injecting new models into the database.

The constraints and indexes were being added too soon, which could cause problems when applying more complicated batches of evolution.

- Fixed issues with setting non-string initial data from a callable.
- Fixed attempting to temporarily remove indexes and constraints that reference models not yet injected into the database.
- Fixed edge cases with the tracking of standard vs. unique indexes in database state on Django 1.6.

## MySQL

- Fixed bad attempts at applying defaults to certain field types.  
Django Evolution will no longer apply a default on `text`, `blob`, `json`, and all `short/medium/long` variations of those.

## Python Compatiblity

- Fixed an unintended deprecation warning with the `collections` module when running on Python 3.10.

## Contributors

- Christian Hammond
- David Trowbridge

## 7.1.3 Django Evolution 2.1.4

**Release date:** February 28, 2022

## Bug Fixes

- Fixed a crash when applying Django compatibility patches on Django < 2.0 when `mysqlclient` isn't installed.

## Contributors

- Christian Hammond
- David Trowbridge

## 7.1.4 Django Evolution 2.1.3

**Release date:** January 25, 2022

## Compatibility Changes

- Patched compatibility between modern versions of `mysqlclient` and Django <= 1.11.  
Django, up through 1.11, attempted to access a `bytes` key in an internal mapping on the database connection handle supplied by `mysqlclient`. This wasn't intended to be present, and was due to a Python 2/3 compatibility issue.  
They worked around this for a while, but dropped that support in the recent 2.1 release. To maintain compatibility, Django Evolution now patches Django's own copy of the mapping table to restore the right behavior.
- Patched Python 3.10+'s `collections` module to include legacy imports when using Django 2.0 or older.  
Django 2.0 and older made use of some imports that no longer exist on Python 3.10. Django Evolution will now bring back this support when running this combination of versions of Django.

### Bug Fixes

- During upgrade, evolutions are no longer applied to newly-added models.
  - Fixed comparison issues between `unique_together` state from very old databases and newer evolutions.
- This could lead to issues applying evolutions that only supply a `unique_together` baseline, or that differ in terms of using tuples or lists.
- Fixed an edge case where the `django_evolution` app could be loaded too early when setting up a new database, causing crashes.
  - Updated to avoid using some deprecated Python and Django functionality.

We had some imports and function calls that were emitting deprecation warnings, depending on the versions of Python and Django. Code has been updated to use modern imports and calls where possible,

### Contributors

- Christian Hammond
- David Trowbridge

## 7.1.5 Django Evolution 2.1.2

**Release date:** January 19, 2021

### Bug Fixes

- Fixed a regression with adding new non-NULL columns on SQLite databases.
- Fixed a possible data loss bug when changing NULL columns to non-NULL on SQLite databases.

### Contributors

- Christian Hammond

## 7.1.6 Django Evolution 2.1.1

**Release date:** January 17, 2021

### Bug Fixes

- Fixed changing a `DecimalField`'s `decimal_places` and `max_digits` attributes.
- Changed the “No upgrade required” text to “No database upgrade required.”

While not a bug, this does help avoid confusion when running as part of a project’s upgrade process, when database changes aren’t the only changes being made.

## Contributors

- Christian Hammond

## 7.1.7 Django Evolution 2.1

**Release date:** November 16, 2020

### New Features

- Dependency management for evolutions and migrations.

Evolutions can now specify other evolutions and migrations that must be applied either before or after. This allows evolutions to, for instance, introduce a model that would be required by another migration (useful for Django apps that have migrations that depend on a swappable model specified in settings).

Django Evolution will determine the correct order in which to apply migrations and evolutions, so as to correctly create or update the database.

Dependencies can be defined per-evolution or per-app. They can depend on specific evolutions or on app evolutions for an app, or on specific migrations.

See [Adding Dependencies](#) for more information.

- Improved transaction management.

Transactions are managed a bit more closely now, allowing more operations to be performed in a transaction at a time and for those operations to be rolled back if anything goes wrong. This should improve reliability of an upgrade.

### Bug Fixes

#### General

- Fixed the order in which models are created.

There was a regression in 2.0 where models could be created in the wrong order, causing issues with applying constraints between those models.

- Fixed error messages in places if stored schema signatures were missing.

Previously, some missing schema signatures could lead to outright crashes, if things went wrong. There's now checks in more places to ensure there's at least a reasonable error message.

### MySQL/MariaDB

- Fixed preserving the `db_index=` values for fields on Django 1.8 through 1.10.

These versions of Django “temporarily” unset the `db_index` attribute on fields when generating SQL for creating indexes, and then never restore it. We now monkey-patch these versions of Django to restore these values.

### Contributors

- Christian Hammond

## 7.1.8 Django Evolution 2.0

**Release date:** August 13, 2020

### New Features

#### All-New Documentation

We have [new documentation](#) for Django Evolution, covering installation, usage, a FAQ, and all release notes.

#### Support for Python 3

Django Evolution is now fully compatible with Python 2.7 and 3.5 through 3.8, allowing it to work across all supported versions of Django.

Speaking of that...

#### Support for Django 1.6 through 3.1

Django Evolution 2.0 supports Django 1.6 through 3.1. Going forward, it will continue to support newer versions of Django as they come out.

This includes modern features, like `Meta.indexes` and `Meta.conditions`.

We can offer this due to the new cooperative support for Django's schema migrations.

### Compatibility with Django Migrations

Historically, Django Evolution has been a standalone schema migration framework, and was stuck with supporting versions of Django prior to 1.7, since evolutions and migrations could not co-exist.

That's been resolved. Django Evolution now controls the entire process, applying both migrations and evolutions together, ensuring a smooth upgrade. Projects get the best of both worlds:

- The ability to use apps that use migrations (most everything, including Django itself)
- Optimized upgrades for the project's own evolution-based models (especially when applying large numbers of evolutions to the same table)

## New Evolve Command

In Django Evolution 2.0, the `evolve` command becomes the sole way of applying both evolutions and migrations, replacing the `migrate`/`syncdb` commands.

To set up or upgrade a database (using both evolutions and migrations), you'll simply run `evolve --execute`. This will work across all versions of Django.

The old `migrate` and `syncdb` commands will still technically work, but they'll wrap `evolve --execute`.

This can all be disabled by setting `DJANGO_EVOLUTION_ENABLED = False` in `settings.py`.

---

**Note:** `initial_data` fixtures will no longer be loaded. These have already been deprecated in Django, but it's worth mentioning for users of older versions of Django.

Also, the `migrate` command will no longer allow individual migrations to be applied.

---

## Moving Apps to Migrations

Projects can transition some or all of their apps to migrations once the last of the evolutions are applied, allowing them to move entirely onto migrations if needed. This is done with the new `MoveToMigrations` mutation.

Simply add one last evolution for an app:

```
from django_evolution.mutations import MoveToDjangoMigrations

MUTATIONS = [
    MoveToDjangoMigrations(),
]
```

This will apply after the last evolution is applied, and from then on all changes to the models will be controlled via migrations.

---

**Note:** Once an app has been moved to migrations, it cannot be moved back to evolutions.

---

## Improved Database Compatibility

- Support for constraints on modern versions of MySQL/MariaDB.

Modern versions of MySQL and MariaDB are now explicitly supported, allowing projects using Django 2.2+ to take advantage of `CHECK` constraints. This requires MySQL 8.0.16+ or MariaDB 10.2.1+ on Django 3.0+.

- Faster and safer SQLite table rebuilds.

Changes to SQLite databases are now optimized, resulting in far fewer table rebuilds when changes are made to a model.

- Support for SQLite 3.25+ column renaming.

SQLite 3.25 introduced `ALTER TABLE ... RENAME COLUMN` syntax, which is faster than a table rebuild and avoids a lot of issues with preserving column references.

- We use Django 1.7's schema rewriting for more of the SQL generation.

This helps ensure future compatibility with new releases of Django, and allows for leveraging more of Django's work toward database compatibility.

### Project-Defined Custom Evolutions

Projects can provide a new `settings.CUSTOM_EVOLUTIONS` setting to define custom evolution modules for apps that don't otherwise make use of evolutions or migrations. The value is a mapping of app module names (same ones you'd see in `settings.INSTALLED_APPS`) to an evolutions module path.

This looks like:

```
CUSTOM_EVOLUTIONS = {  
    'other_project.contrib.foo': 'my_project.compat.foo.evolutions',  
}
```

### Evolver API

The entire evolution/migration process can now be controlled programmatically through the `Evolver` class. This allows an entire database, or just select apps, to be evolved without calling out to a management command.

While most projects will not have a need for this, it's available to those that might want some form of specialized control over the evolution process (for automation, selectively evolving models from an extension/plug-in, or providing an alternative management/upgrade experience).

During an evolution, new signals are emitted, allowing apps to hook into the process and perform any updates they might need:

- `evolved`
- `evolving`
- `evolving_failed`
- `applying_evolution`
- `applied_evolution`
- `applying_migration`
- `applied_migration`
- `created_models`
- `creating_models`

### New Database Signature Format

Django Evolution stores a representation of the database in the `Version` table, in order to track what's been applied and what changes have been made since.

Historically, this has used some older structured data schema serialized in Pickle Protocol 0 format. As of Django Evolution 2.0, it's now using a new schema stored in JSON format, which is designed for future extensibility.

Internally, this is represented by a `set of classes` with a solid API that's independent of the storage format. This eases the addition of new features, and makes it easier to diagnose problems or write custom tools.

**Warning:** This will impact any `SQLMutations` that modify a signature. These will need to be updated to use the new classes, instead of modifying the older schema dictionaries.

## Bug Fixes

### SQLite

- Fixed constraint references from other tables when renaming primary key columns.
- Fixed restoring all table indexes after rebuilding a table.

### Contributors

- Christian Hammond

## 7.2 0.7 Releases

### 7.2.1 Django Evolution 0.7.8

**Release date:** June 14, 2018

## Packaging

- Eggs and wheels are now built only for Python 2.7.  
Older versions of Python are no longer packaged. Source tarballs may work, but we recommend that anyone still on older versions of Python upgrade at their earliest convenience.

## Bug Fixes

- Fixed an issue generating `unique_together` constraints on Postgres in some configurations.  
Depending on the table/index names, `unique_together` constraints could fail to generate on Posrgres, since the names weren't being escaped.

### Contributors

- Christian Hammond

### 7.2.2 Django Evolution 0.7.7

**Release date:** May 25, 2017

#### New Features

- Added a note about backing up the database and not cancelling before executing an evolution.

The confirmation prompt for executing an evolution now suggests backing up the database first. This is only shown in interactive mode.

After the user has confirmed, they're told it may take time and to not cancel the upgrade.

- Added more output when performing evolutions for apps.

When evolving the database, a message is now outputted to the console for each app being evolved. This gives a sense of progress for larger evolutions.

If the evolution fails, an error message will be shown listing the app that failed evolution, the specific SQL statement that failed, and the database error. This can help when diagnosing and recovering from the problem.

- Added an option for writing hinted evolution files.

There's now an `evolve -w--write` option that can be used with `evolve --hint` that writes the hinted evolution to the appropriate directories in the tree. This takes the name that should be used for the evolution file.

This will not update the `evolutions/__init__.py` file.

#### Bug Fixes

- Fixed issues with evolution optimizations when renaming models.

Django Evolution's evolution optimization code had issues when applying a series of evolutions that add a `ForeignKey` field to a newly-introduced model that is then renamed in the same batch. The resulting field would still point to the original model, resulting in a `KeyError`.

#### Contributors

- Christian Hammond

### 7.2.3 Django Evolution 0.7.6

**Release date:** December 1, 2015

#### Bug Fixes

- Fixed a false positive with schema errors when applying evolutions on MySQL.

When applying new evolutions along with baseline schemas for new models, two version history entries are created, one for the new baselines, and one for the new, final schema. On MySQL, this can happen so quickly that they'll end up with the same timestamp (as there isn't a lot of precision in these fields).

Due to internal sort orders, the next evolution then finds the version entry for the baseline schema, and not the final evolved schema, causing it to fail saying that there are changes that couldn't be applied.

This fixes this problem by improving the sorting order.

- Fixed issues evolving certain changes from old database schemas.

Old database schemas didn't track certain information, like the `index_together` information. The code was previously assuming the existence of this information and failing if it wasn't there. Evolving from these older schemas now works.

## Contributors

- Barret Rennie
- Christian Hammond

## 7.2.4 Django Evolution 0.7.5

**Release date:** April 13, 2015

## Bug Fixes

- Mutations on fields with the same name across different models no longer results in conflicts.

With the new optimizer in [Django Evolution 0.7](#), it was possible for mutations to be incorrectly optimized out if, for example, a field was added in one model and then later changed in another model, if both fields had the same name. This was due to the way in which we mapped mutations, and would result in an error in the validation stage before attempting any database modifications. There are no longer any conflicts between same-named field.

- Indexes are no longer created/deleted unnecessarily.

If setting an index for a field, and it already exists in the database, there's no longer an attempt at creating it. Likewise, there's no longer an attempt at deleting an index that does not exist.

## Contributors

- Christian Hammond

## 7.2.5 Django Evolution 0.7.4

**Release date:** September 15, 2014

## New Features

- Add a `RenameModel` mutation for handling model renames.

The new `RenameModel` mutation allows an evolution to indicate that a model has been renamed. This handles updating the signature for any related `ForeignKey` or `ManyToManyField` fields and generating any SQL to perform the table rename (if needed).

## Contributors

- Christian Hammond

## 7.2.6 Django Evolution 0.7.3

**Release date:** July 24, 2014

### Bug Fixes

- Fixed issues evolving `unique_together` attributes on models.

When adding `unique_together` constraints and then changing them within a single evolve operation, any constraints listed more than once would result in unnecessary duplicate SQL statements. These would cause errors that would prevent the transaction from completing.

- Adding and removing a `unique_together` constraint within an evolve operation no longer breaks on PostgreSQL.
- Errors importing a database backend on a modern Django no longer results in unrelated errors about `settings.DATABASE_ENGINE`.

## Contributors

- Christian Hammond

## 7.2.7 Django Evolution 0.7.2

**Release date:** June 2, 2014

### Bug Fixes

- Fixed a crash from no-op column renames on PostgreSQL.

When attempting to rename a column on PostgreSQL and specifying a “new” name that was the same as the old name, the result would be a crash. This is similar to the bug fixed in [Django Evolution 0.7.1](#).

## Contributors

- Christian Hammond

## 7.2.8 Django Evolution 0.7.1

**Release date:** May 21, 2014

## New Features

- Fixed a crash from no-op column renames on MySQL.

When attempting to rename a column on MySQL and specifying a “new” name that was the same as the old name, the result would be a crash. Likewise, there were crashes when renaming a `ManyToManyField`.

## Contributors

- Christian Hammond

## 7.2.9 Django Evolution 0.7

**Release date:** February 3, 2014

## Packaging

- Fixed the unit tests module being accidentally bundled with the package. (Bug #134)
- Fixed the missing NEWS file in the releases. (Bug #130)

## Compatibility Changes

- Added compatibility with Django 1.5 and 1.6 (Bug #136).
- Dropped compatibility for versions of Django prior to 1.4.10.

## New Features

- Added better support for dealing with indexes in the database.

Django changed how index names were generated over time, leading to issues when evolving old databases. We now scan the database prior to evolution, gather the indexes, and look them up based on field data dynamically, guaranteeing we find the correct index.

It's also more resilient now when using custom indexes placed by an administrator.

- Added support for evolving `unique_together` and `index_together` fields.

`unique_together` was previously stored, but ignored, meaning that changes to a `unique_together` would not ever apply to an existing database.

`index_together`, on the other hand, is new in Django 1.5, and was never even stored.

There's now a `ChangeMeta` mutation that allows for changing `unique_together` and `index_together`.

Models making use of `unique_together` or `index_together` will have to supply evolutions defining the current, correct values. These will appear when running `evolve --hint`.

- Optimized the SQL before altering the database.

Mutations are now pre-processed and their output post-processed in order to reduce the number of table-altering mutations. This should massively reduce the amount of time it takes to update a database, particularly when there are multiple `AddField`, `ChangeField`, or `DeleteField` mutations on a single table.

This is the biggest change in this release, and while it's been tested on some large sets of mutations, there may be regressions. Please report any issues you find.

Custom field mutation classes will need to be updated to work with these changes.

### Bug Fixes

- Fixed a number of issues with constraints on different databases. (Bug #127)
- Fixed an invalid variable reference when loading SQL evolution files. (Bug #121)
- SQL evolution files no longer break if there are blank lines. (Bug #111)
- Booleans are now normalized correctly when saving in the database. (Bug #125)  
Previously, invalid boolean values would be used, causing what should have been a “false” value to be “true”.

### Usage

- The evolve command no longer recommends running `evolve --hint --execute`, which can easily cause unwanted problems.

### Testing

- Added easier unit testing for multiple database types.  
The `./tests/runtests.py` script now takes a database type as an argument. The tests will be run against that type of database.  
To make use of this, copy `test_db_settings.py tmpl` to `test_db_settings.py` and fill in the necessary data.
- Fixed all the known unit test failures.
- Rewrote the test suite for better reporting and maintainability.

### Contributors

- Christian Hammond

## 7.2.10 Django Evolution 0.7 Beta 1

**Release date:** January 14, 2014

### Packaging

- Fixed the unit tests module being accidentally bundled with the package. (Bug #134)
- Fixed the missing NEWS file in the releases. (Bug #130)

## Compatibility Changes

- Added compatibility with Django 1.5 (Bug #136).
- Dropped compatibility for versions of Django prior to 1.4.10.

## New Features

- Added better support for dealing with indexes in the database.

Django changed how index names were generated over time, leading to issues when evolving old databases. We now scan the database prior to evolution, gather the indexes, and look them up based on field data dynamically, guaranteeing we find the correct index.

It's also more resilient now when using custom indexes placed by an administrator.

- Added support for evolving `unique_together` and `index_together` fields.

`unique_together` was previously stored, but ignored, meaning that changes to a `unique_together` would not ever apply to an existing database.

`index_together`, on the other hand, is new in Django 1.5, and was never even stored.

There's now a `ChangeMeta` mutation that allows for changing `unique_together` and `index_together`.

Models making use of `unique_together` or `index_together` will have to supply evolutions defining the current, correct values. These will appear when running `evolve --hint`.

- Optimized the SQL before altering the database.

Mutations are now pre-processed and their output post-processed in order to reduce the number of table-altering mutations. This should massively reduce the amount of time it takes to update a database, particularly when there are multiple `AddField`, `ChangeField`, or `DeleteField` mutations on a single table.

This is the biggest change in this release, and while it's been tested on some large sets of mutations, there may be regressions. Please report any issues you find.

Custom field mutation classes will need to be updated to work with these changes.

## Bug Fixes

- Fixed a number of issues with constraints on different databases. (Bug #127)
- Fixed an invalid variable reference when loading SQL evolution files. (Bug #121)
- SQL evolution files no longer break if there are blank lines. (Bug #111)
- Booleans are now normalized correctly when saving in the database. (Bug #125)

Previously, invalid boolean values would be used, causing what should have been a “false” value to be “true”.

### Usage

- The evolve command no longer recommends running `evolve --hint --execute`, which can easily cause unwanted problems.

### Testing

- Added easier unit testing for multiple database types.

The `./tests/runtests.py` script now takes a database type as an argument. The tests will be run against that type of database.

To make use of this, copy `test_db_settings.py.tpl` to `test_db_settings.py` and fill in the necessary data.

- Fixed all the known unit test failures.
- Rewrote the test suite for better reporting and maintainability.

### Contributors

- Christian Hammond

## 7.3 0.6 Releases

### 7.3.1 Django Evolution 0.6.9

**Release date:** March 13, 2013

### Bug Fixes

- Django Evolution no longer applies upgrades that match the current state.

When upgrading an old database, where a new model has been introduced and evolutions were added on that model, Django Evolution would try to apply the mutations after creating that baseline, resulting in confusing errors.

Now we only apply mutations for parts of the database that differ between the last stored signature and the new signature. It should fix a number of problems people have hit when upgrading extremely old databases.

### Contributors

- Christian Hammond

## 7.3.2 Django Evolution 0.6.8

**Release date:** February 8, 2013

### New Features

- Added two new management commands: `list-evolutions` and `wipe-evolution`.

`list-evolutions` lists all applied evolutions. It can take one or more app labels, and will restrict the output to those apps.

`wipe-evolution` will wipe one or more evolutions from the database. This should only be used if absolutely necessary, and can cause problems. It is useful if there's some previously applied evolutions getting in the way, which can happen if a person is uncareful with downgrading and upgrading again.

### Contributors

- Christian Hammond

## 7.3.3 Django Evolution 0.6.7

**Release date:** April 12, 2012

### Bug Fixes

- Don't fail when an app doesn't contain any models.

Installing a baseline for apps without models was failing. The code to install a baseline evolution assumed that all installed apps would have models defined, but this wasn't always true. We now handle this case and just skip over such apps.

### Contributors

- Christian Hammond

## 7.3.4 Django Evolution 0.6.6

**Release date:** April 1, 2012

### New Features

- Generate more accurate sample evolutions.

The sample evolutions generated with `evolve --hint` should now properly take into account import paths for third-party database modules. Prior to this, such an evolution had to be modified by hand to work.

- Generate PEP-8-compliant sample evolutions.

The evolutions are now generated according to the standards of PEP-8. This mainly influences blank lines around imports and the grouping of imports.

- Support Django 1.4's timezone awareness in the `Version` model.

The `Version` model was generating runtime warnings when creating an instance of the model under Django 1.4, due to using a naive (non-timezone-aware) datetime. We now try to use Django's functionality for this, and fall back on the older methods for older versions of Django.

### Contributors

- Christian Hammond

## 7.3.5 Django Evolution 0.6.5

**Release date:** August 15, 2011

### New Features

- Added a built-in evolution to remove the Message model in Django 1.4 SVN.

Django 1.4 SVN removes the `Message` model from `django.contrib.auth`. This would break evolutions, since there wasn't an evolution for this. We now install one if we detect that the `Message` model is gone.

### Bug Fixes

- Fixed the version association for baseline evolutions for apps.

The new code for installing a baseline evolution for new apps in *Django Evolution 0.6.4* was associating the wrong `Version` model with the `Evolution`. This doesn't appear to cause any real-world problems, but it does make it harder to see the proper evolution history in the database.

### Contributors

- Christian Hammond

## 7.3.6 Django Evolution 0.6.4

**Release date:** June 22, 2011

### New Features

- Install a baseline evolution history for any new apps.

When upgrading an older database using Django Evolution when a new model has been added and subsequent evolutions were made on that model, the upgrade would fail. It would attempt to apply those evolutions on that model, which, being newly created, would already have those new field changes.

Now, like with an initial database, we install a baseline evolution history for any new apps. This will ensure that those evolutions aren't applied to the models in that app.

## Bug Fixes

- Fixed compatibility with Django SVN in the unit tests.

In Django SVN r16053, `get_model()` and `get_models()` only return installed modules by default. This is calculated in part by a new `AppCache.app_labels` dictionary, along with an existing `AppCache.app_store`, neither of which we properly populated.

We now set both of these (though, `app_labels` only on versions of Django that have it). This allows the unit tests to pass, both with older versions of Django and Django SVN.

## Contributors

- Christian Hammond

## 7.3.7 Django Evolution 0.6.3

**Release date:** May 9, 2011

## Bug Fixes

- Fixed multi-database support with different database backends.

The multi-database support only worked when the database backends matched. Now it should work with different types. The unit tests have been verified to work now with different types of databases.

- Fixed a breaking with PostgreSQL when adding non-null columns with default values. (Bugs #58 and #74)

Adding new columns that are non-null and have a default value would break with PostgreSQL when the table otherwise had data in it. The SQL for adding a column is an `ALTER TABLE` followed by an `UPDATE` to set all existing records to have the new default value. PostgreSQL, however, doesn't allow this within the same transaction.

Now we use two `ALTER TABLEs`. The first adds the column with a default value, which should affect existing records. The second drops the default. This should ensure that the tables have the data we expect while at the same time keeping the field attributes the same as what Django would generate.

## Contributors

- Christian Hammond

## 7.3.8 Django Evolution 0.6.2

**Release date:** November 19, 2010

### New Features

- Add compatibility with Django 1.3.

Django 1.3 introduced a change to the `Session.expire_date` field's schema, setting `db_index` to `True`. This caused Django Evolution to fail during evolution, with no way to provide an evolution file to work around the problem. Django Evolution now handles this by providing the evolution when running with Django 1.3 or higher.

### Contributors

- Christian Hammond

## 7.3.9 Django Evolution 0.6.1

**Release date:** October 25, 2010

### Bug Fixes

- Fixed compatibility problems with both Django 1.1 and Python 2.4.

### Contributors

- Christian Hammond

## 7.3.10 Django Evolution 0.6

**Release date:** October 24, 2010

### New Features

- Added support for Django 1.2's ability to use multiple databases.

This should use the existing routers used in your project. By default, operations will happen on the 'default' database. This can be overridden during evolution by passing `--database=<dbname>` to the `evolve` command.

Patch by Marc Bee and myself.

### Contributors

- Christian Hammond
- Marc Bee

## 7.4 0.5 Releases

### 7.4.1 Django Evolution 0.5.1

**Release date:** October 13, 2010

#### New Features

- Made the `evolve` management command raise `CommandError` instead of `sys.exit()` on failure. This makes it callable from third party software.

Patch by Mike Conley.

- Made the `evolve` functionality available through an `evolve()` function in the management command, allowing the rest of the command-specific logic to be skipped (such as console output and prompting).

Patch by Mike Conley.

#### Bug Fixes

- Fixed incorrect defaults on SQLite when adding null fields. (Bug #49)

On SQLite, adding a null field without a default value would cause the field name to be the default. This was due to attempting to select the field name from the temporary table, but since the table didn't exist, the field name itself was being used as the value.

We are now more explicit about the fields being selected and populated. We have two lists, and no longer assume both are identical. We also use NULL columns for temporary table fields unconditionally.

Patch by myself and Chris Beaven.

#### Contributors

- Chris Beaven
- Christian Hammond
- Mike Conley

### 7.4.2 Django Evolution 0.5

**Release date:** May 18, 2010

Initial public release.



## DJANGO EVOLUTION DOCUMENTATION

Django Evolution is a database schema migration tool for projects using the [Django](#) web framework. Its job is to help projects make changes to a database's schema – the structure of the tables and columns and indexes – in the fastest way possible (incurring minimum downtime) and in a way that works across all Django-supported databases.

This is very similar in concept to the built-in [\*migrations\*](#) support in Django 1.7 and higher. Django Evolution predates both Django's own migrations, and works alongside it to transition databases taking advantage of the strengths of both migrations and evolutions.

While most will be fine with [\*migrations\*](#), there's a couple reasons why you might find Django Evolution a worthwhile addition to your project:

1. You're still stuck on Django 1.6 or earlier and need to make changes to your database.

Django 1.6 is the last version without built-in support for migrations, and there are still codebases out there using it. Django Evolution can help keep upgrades manageable, and make it easier to transition all or part of your codebase to migrations when you finally upgrade.

2. You're distributing a self-installable web application, possibly used in large enterprises, where you have no control over when people are going to upgrade.

Django's migrations assume some level of planning around when changes are made to the schema and when they're applied to a database. The more changes you make, and the more versions in-between what the user is running and what they upgrade to, the longer the upgrade time.

If a customer is in control of when they upgrade, they might end up with *years* of migrations that need to be applied.

Migrations apply one-by-one, possibly triggering the rebuild of a table many times during an upgrade. Django Evolution, on the other hand, can apply years worth of evolutions at once, optimized to perform as few table changes as possible. This can take days, hours or even *seconds* off the upgrade time.

Django Evolution officially supports Django 1.6 through 4.2.

### 8.1 Questions So Far?

- *How Does It Work?*

## 8.2 Let's Get Started

- *Install Django Evolution*
- *Writing Your First Evolution*
- *Exploring App and Model Mutations*
- *Apply evolutions with `evolve --execute`*

## 8.3 Reference

### 8.3.1 Project Versioning Policy

Beginning with 2.0, Django Evolution uses semantic versioning, in `major.minor.micro` form.

We will bump `major` any time there is a backwards-incompatible change to:

- Evolution definition format
- Compatibility with older versions of Django, Python, or databases
- The `evolve` management command's arguments or behavior
- *Public Python API*

We will bump `minor` any time there's a new feature.

We will bump `micro` any time there's just bug or packaging fixes.

### 8.3.2 Module and Class References

---

**Note:** Most of the codebase should not be considered stable API, as many parts will change.

The code documented here is a subset of the codebase. Backend database implementations and some internal modules are not included.

---

## Public API

<code>django_evolution</code>	Django Evolution version and package information.
<code>django_evolution.conf</code>	Configuration for Django Evolution.
<code>django_evolution.consts</code>	Constants used throughout Django Evolution.
<code>django_evolution.deprecation</code>	Internal support for handling deprecations in Django Evolution.
<code>django_evolution.errors</code>	Standard exceptions for Django Evolution.
<code>django_evolution.evolve</code>	Main interface for evolving applications.
<code>django_evolution.evolve.base</code>	Base classes for evolver-related objects.
<code>django_evolution.evolve.evolver</code>	Main Evolver interface for performing evolutions and migrations.
<code>django_evolution.evolve.evolve_app_task</code>	Task for evolving an application.
<code>django_evolution.evolve.purge_app_task</code>	Task for purging an application.
<code>django_evolution.models</code>	Database models for tracking project schema history.
<code>django_evolution.mutations</code>	Mutations for models, fields, and applications.
<code>django_evolution.mutations.add_field</code>	Mutation that adds a field to a model.
<code>django_evolution.mutations.base</code>	Base support for mutations.
<code>django_evolution.mutations.change_field</code>	Mutation that changes attributes on a field.
<code>django_evolution.mutations.change_meta</code>	Mutation that changes meta properties on a model.
<code>django_evolution.mutations.delete_application</code>	Mutation that deletes an application.
<code>django_evolution.mutations.delete_field</code>	Mutation for deleting fields from a model.
<code>django_evolution.mutations.delete_model</code>	Mutation that deletes a model.
<code>django_evolution.mutations.move_to_django_migrations</code>	Mutation that moves an app to Django migrations.
<code>django_evolution.mutations.rename_app_label</code>	Mutation that renames the app label for an application.
<code>django_evolution.mutations.rename_field</code>	Mutation that renames a field on a model.
<code>django_evolution.mutations.rename_model</code>	Mutation that renames a model.
<code>django_evolution.mutations.sql_mutation</code>	Mutation for executing SQL statements.
<code>django_evolution.serialization</code>	Serialization and deserialization.
<code>django_evolution.signals</code>	Signals for monitoring the evolution process.
<code>django_evolution.signature</code>	Classes for working with stored evolution state signatures.

### `django_evolution`

Django Evolution version and package information.

These variables and functions can be used to identify the version of Review Board. They're largely used for packaging purposes.

### Functions

---

```
get_package_version()
```

---

```
get_version_string()
```

---

```
is_release()
```

---

```
django_evolution.get_version_string()
```

```
django_evolution.get_package_version()
```

```
django_evolution.is_release()
```

### **django\_evolution.conf**

Configuration for Django Evolution.

New in version 2.2.

### Classes

---

<i>DjangoEvolutionSettings</i> (settings_module)	Settings for Django Evolution.
--	--------------------------------

---

```
class django_evolution.conf.DjangoEvolutionSettings(settings_module)
```

Bases: `object`

Settings for Django Evolution.

This wraps the settings defined in `django.conf.settings`. If `settings.DJANGO_EVOLUTION` is set, then all supported keys will be loaded.

Legacy settings ( `settings.DJANGO_EVOLUTION_ENABLED` and `settings.CUSTOM_EVOLUTIONS`), if found, will be loaded, and will cause a deprecation warning to be emitted.

New in version 2.2.

#### **CUSTOM\_EVOLUTIONS**

A mapping of app labels to lists of custom evolution modules.

Type

`dict`

#### **ENABLED**

Whether Django Evolution is enabled.

If enabled, the `syncdb` and `migrate` management commands will instead use Django Evolution. Post-syncdb/migrate operations will also cause Django Evolution to track state.

If disabled, the management commands will operate no differently than in a normal Django installation.

Type

`bool`

**`__init__(settings_module)`**

Initialize the settings wrapper.

**Parameters**

**settings\_module** (`module`) – The Django settings module to load from.

**`load_settings(settings_module)`**

Set defaults and load settings.

**Parameters**

**settings\_module** (`module`) – The Django settings module to load from.

**`replace_settings(new_settings)`**

Replace settings from a dictionary.

This is expected to take the equivalent of a `settings.DJANGO_EVOLUTION` dictionary. Any valid settings found will be loaded. Any not found will be set back to defaults.

**Parameters**

**new\_settings** (`dict`) – The new settings dictionary.

**`django_evolution.consts`**

Constants used throughout Django Evolution.

**Classes**

<code>EvolutionsSource()</code>	The source for an app's evolutions.
<code>UpgradeMethod()</code>	Upgrade methods available for an application.

**`class django_evolution.consts.UpgradeMethod`**

Bases: `object`

Upgrade methods available for an application.

**`EVOLUTIONS = 'evolutions'`**

The app is upgraded through Django Evolution.

**`MIGRATIONS = 'migrations'`**

The app is upgraded through Django Migrations.

**`class django_evolution.consts.EvolutionsSource`**

Bases: `object`

The source for an app's evolutions.

**`APP = 'app'`**

The evolutions are provided by the app.

**`BUILTIN = 'builtin'`**

The evolutions are built-in to Django Evolution.

**`PROJECT = 'project'`**

The evolutions are provided custom by the project.

### `django_evolution.deprecation`

Internal support for handling deprecations in Django Evolution.

The version-specific objects in this module are not considered stable between releases, and may be removed at any point. The base objects are considered stable.

New in version 2.2.

#### Module Attributes

<code>RemovedInNextDjangoEvolutionWarning</code>	Alias for deprecations in the next Django Evolution release.
--	--

#### Exceptions

<code>BaseRemovedInDjangoEvolutionWarning</code>	Base class for a Django Evolution deprecation warning.
<code>RemovedInDjangoEvolution30Warning</code>	Deprecations for features being removed in Django Evolution 3.0.
<code>RemovedInDjangoEvolution40Warning</code>	Deprecations for features being removed in Django Evolution 4.0.
<code>RemovedInNextDjangoEvolutionWarning</code>	Alias for deprecations in the next Django Evolution release.

#### `exception django_evolution.deprecation.BaseRemovedInDjangoEvolutionWarning`

Bases: `DeprecationWarning`

Base class for a Django Evolution deprecation warning.

All version-specific deprecation warnings inherit from this, allowing callers to check for Django Evolution deprecations without being tied to a specific version.

New in version 2.2.

##### `classmethod warn(message, stacklevel=2)`

Emit the deprecation warning.

This is a convenience function that emits a deprecation warning using this class, with a suitable default stack level. Callers can provide a useful message and a custom stack level.

##### Parameters

- `message` (`unicode`) – The message to show in the deprecation warning.
- `stacklevel` (`int`, *optional*) – The stack level for the warning.

#### `exception django_evolution.deprecation.RemovedInDjangoEvolution30Warning`

Bases: `BaseRemovedInDjangoEvolutionWarning`

Deprecations for features being removed in Django Evolution 3.0.

Note that this class will itself be removed in Django Evolution 3.0. If you need to check against Django Evolution deprecation warnings, please see `BaseRemovedInDjangoEvolutionWarning`.

New in version 2.2.

**exception django\_evolution.deprecation.RemovedInDjangoEvolution40Warning**Bases: *BaseRemovedInDjangoEvolutionWarning*

Deprecations for features being removed in Django Evolution 4.0.

Note that this class will itself be removed in Django Evolution 4.0. If you need to check against Django Evolution deprecation warnings, please see *BaseRemovedInDjangoEvolutionWarning*. Alternatively, you can use the alias for this class, *RemovedInNextDjangoEvolutionWarning*.

New in version 2.2.

**django\_evolution.deprecation.RemovedInNextDjangoEvolutionWarning**

Alias for deprecations in the next Django Evolution release.

**django\_evolution.errors**

Standard exceptions for Django Evolution.

**Exceptions**

<i>BaseMigrationError</i> (msg)	Base class for migration errors.
<i>CannotSimulate</i> (msg)	A mutation cannot be simulated.
<i>DatabaseStateError</i> (msg)	There was an issue working with database state.
<i>DjangoEvolutionSupportError</i> (msg)	A feature isn't supported by the current version of Django.
<i>EvolutionBaselineMissingError</i> (msg)	An evolution baseline is missing.
<i>EvolutionException</i> (msg)	Base class for a Django Evolution exception.
<i>EvolutionExecutionError</i> (msg[, app_label, ...])	Execution of an evolution failed.
<i>EvolutionNotImplementedError</i> (msg)	An operation is not supported by the mutation or database backend.
<i>EvolutionTaskAlreadyQueuedError</i> (msg)	The task has already been queued on the evolver.
<i>InvalidSignatureVersion</i> (version)	An invalid signature version was provided or found.
<i>MigrationConflictsError</i> (conflicts)	There are conflicts between migrations.
<i>MigrationHistoryError</i> (msg)	An error with the stored history of migrations.
<i>MissingSignatureError</i> (msg)	A requested signature could not be found.
<i>QueueEvolverTaskError</i> (msg)	Error queueing an evolver task.
<i>SimulationFailure</i> (msg)	A mutation simulation has failed.

**exception django\_evolution.errors.EvolutionException(msg)**Bases: *Exception*

Base class for a Django Evolution exception.

**\_\_init\_\_(msg)****\_\_str\_\_()**

Return str(self).

**exception django\_evolution.errors.EvolutionExecutionError(msg, app\_label=None, detailed\_error=None, last\_sql\_statement=None)**Bases: *EvolutionException*

Execution of an evolution failed.

Details about the failure, including the app that failed and the last SQL statement executed, are available in the exception as attributes.

### **app\_label**

The label of the app that failed evolution. This may be `None`.

#### Type

`unicode`

### **detailed\_error**

Detailed error information from the failure that triggered this exception. This might be another exception's error message, or it may be `None`.

#### Type

`unicode`

### **last\_sql\_statement**

The last SQL statement that was executed. This may be `None`.

#### Type

`unicode`

### **`__init__(msg, app_label=None, detailed_error=None, last_sql_statement=None)`**

Initialize the error.

#### Parameters

- **msg** (`unicode`) – The error message.
- **app\_label** (`unicode`, *optional*) – The label of the app that failed evolution.
- **detailed\_error** (`unicode`, *optional*) – Detailed error information from the failure that triggered this exception. This might be another exception's error message.
- **last\_sql\_statement** (`unicode`, *optional*) – The last SQL statement that was executed.

### **exception django\_evolution.errors.CannotSimulate(msg)**

Bases: `EvolutionException`

A mutation cannot be simulated.

### **exception django\_evolution.errors.SimulationFailure(msg)**

Bases: `EvolutionException`

A mutation simulation has failed.

### **exception django\_evolution.errors.EvolutionNotImplementedError(msg)**

Bases: `EvolutionException, NotImplementedError`

An operation is not supported by the mutation or database backend.

### **exception django\_evolution.errors.DatabaseStateError(msg)**

Bases: `EvolutionException`

There was an issue working with database state.

### **exception django\_evolution.errors.MissingSignatureError(msg)**

Bases: `EvolutionException`

A requested signature could not be found.

```
exception django_evolution.errors.QueueEvolverTaskError(msg)
```

Bases: *EvolutionException*

Error queueing an evolver task.

```
exception django_evolution.errors.EvolutionTaskAlreadyQueuedError(msg)
```

Bases: *QueueEvolverTaskError*

The task has already been queued on the evolver.

```
exception django_evolution.errors.EvolutionBaselineMissingError(msg)
```

Bases: *EvolutionException*

An evolution baseline is missing.

```
exception django_evolution.errors.InvalidSignatureVersion(version)
```

Bases: *EvolutionException*

An invalid signature version was provided or found.

```
__init__(version)
```

Initialize the exception.

**Parameters**

**version** (`int`) – The invalid signature version.

```
exception django_evolution.errors.BaseMigrationError(msg)
```

Bases: *EvolutionException*

Base class for migration errors.

```
exception django_evolution.errors.MigrationHistoryError(msg)
```

Bases: *BaseMigrationError*

An error with the stored history of migrations.

This is raised if any applied migrations have unapplied dependencies.

```
exception django_evolution.errors.MigrationConflictsError(conflicts)
```

Bases: *BaseMigrationError*

There are conflicts between migrations.

```
__init__(conflicts)
```

Initialize the error.

**Parameters**

**conflicts** (`dict`) – A dictionary of conflicts, provided by the migrations system.

```
exception django_evolution.errors.DjangoEvolutionSupportError(msg)
```

Bases: *EvolutionException*

A feature isn't supported by the current version of Django.

### `django_evolution.evolve`

Main interface for evolving applications.

Changed in version 2.2: The classes have all moved to nested modules, but this module will continue to provide forwarding imports.

<code>BaseEvolutionTask</code>	Base class for a task to perform during evolution.
<code>Evolver</code>	The main class for managing database evolutions.
<code>EvolveAppTask</code>	A task for evolving models in an application.
<code>PurgeAppTask</code>	A task for purging an application's tables from the database.

### `django_evolution.evolve.base`

Base classes for evolver-related objects.

New in version 2.2: This was previously located in `django_evolution.evolve`.

#### Classes

<code>BaseEvolutionTask(task_id, evolver)</code>	Base class for a task to perform during evolution.
--	--

`class django_evolution.evolve.base.BaseEvolutionTask(task_id, evolver)`

Bases: `object`

Base class for a task to perform during evolution.

##### `can_simulate`

Whether the task can be simulated without requiring additional information.

This is set after calling `prepare()`.

##### Type

`bool`

##### `evolution_required`

Whether an evolution is required by this task.

This is set after calling `prepare()`.

##### Type

`bool`

##### `evolver`

The evolver that will execute the task.

##### Type

`Evolver`

##### `id`

The unique ID for the task.

##### Type

`unicode`

**new\_evolutions**

A list of evolution model entries this task would create.

This is set after calling [prepare\(\)](#).

**Type**

`list of django_evolution.models.Evolution`

**sql**

A list of SQL statements to perform for the task. Each entry can be a string or tuple accepted by [run\\_sql\(\)](#).

**Type**

`list`

**classmethod prepare\_tasks(evolver, tasks, \*\*kwargs)**

Prepare a list of tasks.

This is responsible for calling [prepare\(\)](#) on each of the provided tasks. It can augment this by calculating any other state needed in order to influence the tasks or react to them.

If this applies state to the class, it should always be careful to completely reset the state on each run, in case there are multiple Evolver instances at work within a process.

**Parameters**

- **evolver** (`Evolver`) – The evolver that's handling the tasks.
- **tasks** (`list of BaseEvolutionTask`) – The list of tasks to prepare. These will match the current class.
- **\*\*kwargs** (`dict`) – Keyword arguments to pass to the tasks' `:py:meth:`prepare`` methods.

**classmethod execute\_tasks(evolver, tasks, \*\*kwargs)**

Execute a list of tasks.

This is responsible for calling [execute\(\)](#) on each of the provided tasks. It can augment this by executing any steps before or after the tasks.

If this applies state to the class, it should always be careful to completely reset the state on each run, in case there are multiple Evolver instances at work within a process.

This may depend on state from [prepare\\_tasks\(\)](#).

**Parameters**

- **evolver** (`Evolver`) – The evolver that's handling the tasks.
- **tasks** (`list of BaseEvolutionTask`) – The list of tasks to execute. These will match the current class.
- **\*\*kwargs** (`dict`) – Keyword arguments to pass to the tasks' `:py:meth:`execute`` methods.

**\_\_init\_\_(task\_id, evolver)**

Initialize the task.

**Parameters**

- **task\_id** (`unicode`) – The unique ID for the task.
- **evolver** (`Evolver`) – The evolver that will execute the task.

### `is_mutation Mutable`(*mutation*, *\*\*kwargs*)

Return whether a mutation is mutable.

This is a handy wrapper around `BaseMutation.is_mutable` that passes standard arguments based on evolver state. Callers should pass any additional arguments that are required as keyword arguments.

#### Parameters

- **`mutation`** (`django_evolution.mutations.BaseMutation`) – The mutation to check.
- **`**kwargs (dict)`** – Additional keyword arguments to pass to `BaseMutation.is_mutable`.

#### Returns

True if the mutation is mutable. False if it is not.

#### Return type

`bool`

### `prepare`(*hinted*, *\*\*kwargs*)

Prepare state for this task.

This is responsible for determining whether the task applies to the database. It must set `evolution_required`, `new_evolutions`, and `sql`.

This must be called before `execute()` or `get_evolution_content()`.

#### Parameters

- **`hinted (bool)`** – Whether to prepare the task for hinted evolutions.
- **`**kwargs (dict, unused)`** – Additional keyword arguments passed for task preparation. This is provide for future expansion purposes.

### `execute`(*cursor=None*, *sql\_executor=None*, *\*\*kwargs*)

Execute the task.

This will make any changes necessary to the database.

Changed in version 2.1: `cursor` is now deprecated in favor of `sql_executor`.

#### Parameters

- **`cursor (django.db.backends.util.CursorWrapper, optional)`** – The legacy database cursor used to execute queries.
- **`sql_executor (django_evolution.utils.sql.SQLExecutor, optional)`** – The SQL executor used to run any SQL on the database.
- **`**kwargs (dict)`** – Additional keyword arguments, for future expansion.

#### Raises

`django_evolution.errors.EvolutionExecutionError` – The evolution task failed. Details are in the error.

### `get_evolution_content()`

Return the content for an evolution file for this task.

#### Returns

The evolution content.

#### Return type

`unicode`

**`__repr__()`**

Return a string representation of the task.

**Returns**

The string representation.

**Return type**

`unicode`

**`__str__()`**

Return a string description of the task.

**Returns**

The string description.

**Return type**

`unicode`

## **`django_evolution.evolve.evolver`**

Main Evolver interface for performing evolutions and migrations.

New in version 2.2: This was previously located in `django_evolution.evolve`.

### Classes

---

<code>Evolver([hinted, verbosity, interactive, ...])</code>	The main class for managing database evolutions.
---	--

---

`class django_evolution.evolve.evolver.Evolver(hinted=False, verbosity=0, interactive=False, database_name='default')`

Bases: `object`

The main class for managing database evolutions.

The evolver is used to queue up tasks that modify the database. These allow for evolving database models and purging applications across an entire Django project or only for specific applications. Custom tasks can even be written by an application if very specific database operations need to be made outside of what's available in an evolution.

Tasks are executed in order, but batched by the task type. That is, if two instances of `TaskType1` are queued, followed by an instance of `TaskType2`, and another of `TaskType1`, all 3 tasks of `TaskType1` will be executed at once, with the `TaskType2` task following.

Callers are expected to create an instance and queue up one or more tasks. Once all tasks are queued, the changes can be made using `evolve()`. Alternatively, evolution hints can be generated using `generate_hints()`.

Projects will generally utilize this through the existing `evolve` Django management command.

#### **`connection`**

The database connection object being used for the evolver.

**Type**

`django.db.backends.base.BaseDatabaseWrapper`

**database\_name**

The name of the database being evolved.

**Type**

`unicode`

**database\_state**

The state of the database, for evolution purposes.

**Type**

`django_evolution.db.state.DatabaseState`

**evolved**

Whether the evolver has already performed its evolutions. These can only be done once per evolver.

**Type**

`bool`

**hinted**

Whether the evolver is operating against hinted evolutions. This may result in changes to the database without there being any accompanying evolution files backing those changes.

**Type**

`bool`

**interactive**

Whether the evolution operations are being performed in a way that allows interactivity on the command line. This is passed along to signal emissions.

**Type**

`bool`

**initial\_diff**

The initial diff between the stored project signature and the current project signature.

**Type**

`django_evolution.diff.Diff`

**project\_sig**

The project signature. This will start off as the previous signature stored in the database, but will be modified when mutations are simulated.

**Type**

`django_evolution.signature.ProjectSignature`

**verbosity**

The verbosity level for any output. This is passed along to signal emissions.

**Type**

`int`

**version**

The project version entry saved as the result of any evolution operations. This contains the current version of the project signature. It may be `None` until `evolve()` is called.

**Type**

`django_evolution.models.Version`

`__init__(hinted=False, verbosity=0, interactive=False, database_name='default')`

Initialize the evolver.

## Parameters

- **hinted** (`bool`, *optional*) – Whether to operate against hinted evolutions. This may result in changes to the database without there being any accompanying evolution files backing those changes.
- **verbosity** (`int`, *optional*) – The verbosity level for any output. This is passed along to signal emissions.
- **interactive** (`bool`, *optional*) – Whether the evolution operations are being performed in a way that allows interactivity on the command line. This is passed along to signal emissions.
- **database\_name** (`unicode`, *optional*) – The name of the database to evolve.

## Raises

`django_evolution.errors.EvolutionBaselineMissingError` – An initial baseline for the project was not yet installed. This is due to `syncdb/migrate` not having been run.

## `property tasks`

A list of all tasks that will be performed.

This can only be accessed after all necessary tasks have been queued.

## `can_simulate()`

Return whether all queued tasks can be simulated.

If any tasks cannot be simulated (for instance, a hinted evolution requiring manually-entered values), then this will return `False`.

This can only be called after all tasks have been queued.

### Returns

`True` if all queued tasks can be simulated. `False` if any cannot.

#### Return type

`bool`

## `get_evolution_required()`

Return whether there are any evolutions required.

This can only be called after all tasks have been queued.

### Returns

`True` if any tasks require evolution. `False` if none do.

#### Return type

`bool`

## `diff_evolutions()`

Return a diff between stored and post-evolution project signatures.

This will run through all queued tasks, preparing them and simulating their changes. The returned diff will represent the changes made in those tasks.

This can only be called after all tasks have been queued.

### Returns

The diff between the stored signature and the queued changes.

#### Return type

`django_evolution.diff.Diff`

### `iter_evolution_content()`

Generate the evolution content for all queued tasks.

This will loop through each tasks and yield any evolution content provided.

This can only be called after all tasks have been queued.

#### Raises

`tuple` – A tuple of (`task`, `evolution_content`).

### `queue_evolve_all_apps()`

Queue an evolution of all registered Django apps.

This cannot be used if `queue_evolve_app()` is also being used.

#### Raises

- `djangoevolution.errors.EvolutionTaskAlreadyQueuedError` – An evolution for an app was already queued.
- `djangoevolution.errors.QueueEvolverTaskError` – Error queueing a non-duplicate task. Tasks may have already been prepared and finalized.

### `queue_evolve_app(app)`

Queue an evolution of a registered Django app.

#### Parameters

`app` (`module`) – The Django app to queue an evolution for.

#### Raises

- `djangoevolution.errors.EvolutionTaskAlreadyQueuedError` – An evolution for this app was already queued.
- `djangoevolution.errors.QueueEvolverTaskError` – Error queueing a non-duplicate task. Tasks may have already been prepared and finalized.

### `queue_purge_old_apps()`

Queue the purging of all old, stale Django apps.

This will purge any apps that exist in the stored project signature but that are no longer registered in Django.

This generally should not be used if `queue_purge_app()` is also being used.

#### Raises

- `djangoevolution.errors.EvolutionTaskAlreadyQueuedError` – A purge of an app was already queued.
- `djangoevolution.errors.QueueEvolverTaskError` – Error queueing a non-duplicate task. Tasks may have already been prepared and finalized.

### `queue_purge_app(app_label)`

Queue the purging of a Django app.

#### Parameters

`app_label` (`unicode`) – The label of the app to purge.

#### Raises

- `djangoevolution.errors.EvolutionTaskAlreadyQueuedError` – A purge of this app was already queued.
- `djangoevolution.errors.QueueEvolverTaskError` – Error queueing a non-duplicate task. Tasks may have already been prepared and finalized.

**queue\_task(task)**

Queue a task to run during evolution.

This should only be directly called if working with custom tasks. Otherwise, use a more specific queue method.

**Parameters**

**task** (BaseEvolutionTask) – The task to queue.

**Raises**

- **django\_evolution.errors.EvolutionTaskAlreadyQueuedError** – A purge of this app was already queued.
- **django\_evolution.errors.QueueEvolverTaskError** – Error queueing a non-duplicate task. Tasks may have already been prepared and finalized.

**evolve()**

Perform the evolution.

This will run through all queued tasks and attempt to apply them in a database transaction, tracking each new batch of evolutions as the tasks finish.

This can only be called once per evolver instance.

**Raises**

- **django\_evolution.errors.EvolutionException** – Something went wrong during the evolution process. Details are in the error message. Note that a more specific exception may be raised.
- **django\_evolution.errors.EvolutionExecutionError** – A specific evolution task failed. Details are in the error.

**sql\_executor(\*\*kwargs)**

Return an SQLExecutor for executing SQL.

This is a convenience method for creating an *SQLExecutor* to operate using the evolver's current database.

New in version 2.1.

**Parameters**

**\*\*kwargs** (`dict`) – Additional keyword arguments used to construct the executor.

**Returns**

The new SQLExecutor.

**Return type**

`django_evolution.utils.sql.SQLExecutor`

**transaction()**

Execute database operations in a transaction.

This is a convenience method for executing in a transaction using the evolver's current database.

Deprecated since version 2.1: This has been replaced with manual calls to *SQLExecutor*.

**Context**

`django.db.backends.util.CursorWrapper` – The cursor used to execute statements.

### `django_evolution.evolve.evolve_app_task`

Task for evolving an application.

New in version 2.2: This was previously located in `django_evolution.evolve`.

#### Classes

---

<code>EvolveAppTask(evolver, app[, evolutions, ...])</code>	A task for evolving models in an application.
---	---

---

`class django_evolution.evolve.evolve_app_task.EvolveAppTask(evolver, app, evolutions=None, migrations=None)`

Bases: `BaseEvolutionTask`

A task for evolving models in an application.

This task will run through any evolutions in the provided application and handle applying each of those evolutions that haven't yet been applied.

#### `app`

The app module to evolve.

##### Type

`module`

#### `app_label`

The app label for the app to evolve.

##### Type

`unicode`

`classmethod prepare_tasks(evolver, tasks, hinted=False, **kwargs)`

Prepare a list of tasks.

If migrations are supported, then before preparing any of the tasks, this will begin setting up state needed to apply any migrations for apps that use them (or will use them after any evolutions are applied).

After tasks are prepared, this will apply any migrations that need to be applied, updating the app's signature appropriately and recording all applied migrations.

#### Parameters

- `evolver` (`Evolver`) – The evolver that's handling the tasks.
- `tasks` (`list` of `BaseEvolutionTask`) – The list of tasks to prepare. These will match the current class.
- `**kwargs` (`dict`) – Keyword arguments to pass to the tasks' `:py:meth:`prepare`` methods.

#### Raises

`django_evolution.errors.BaseMigrationError` – There was an error with the setup or validation of migrations. A subclass containing additional details will be raised.

`classmethod execute_tasks(evolver, tasks, **kwargs)`

Execute a list of tasks.

This is responsible for calling `execute()` on each of the provided tasks. It can augment this by executing any steps before or after the tasks.

#### Parameters

- **evolver** (`Evolver`) – The evolver that's handling the tasks.
- **tasks** (`list` of `BaseEvolutionTask`) – The list of tasks to execute. These will match the current class.
- **cursor** (`django.db.backends.util.CursorWrapper`) – The database cursor used to execute queries.
- **\*\*kwargs** (`dict`) – Keyword arguments to pass to the tasks' `:py:meth:`execute`` methods.

### `__init__(evolver, app, evolutions=None, migrations=None)`

Initialize the task.

#### Parameters

- **evolver** (`Evolver`) – The evolver that will execute the task.
- **app** (`module`) – The app module to evolve.
- **evolutions** (`list` of `dict`, *optional*) – Optional evolutions to use for the app instead of loading from a file. This is intended for testing purposes.  
Each dictionary needs a `label` key for the evolution label and a `mutations` key for a list of `BaseMutation` instances.
- **migrations** (`list` of `django.db.migrations.Migration`, *optional*) – Optional migrations to use for the app instead of loading from files. This is intended for testing purposes.

### `generate_mutations_info(pending_mutations, update_evolver=True)`

Generate information on a series of mutations.

This will optimize and run the list of pending mutations against the evolver's stored signature and return the optimized list of mutations and SQL, along with some information on the app.

The evolver's signature will be updated by default, but this can be disabled in order to just retrieve information without making any changes.

#### Parameters

- **pending\_mutations** (`list` of `:class:`BaseMutation``) – The list of pending mutations to run.
- **update\_evolver** (`bool`, *optional*) – Whether to update the evolver's signature.

#### Returns

The resulting information from running the mutations. This includes the following:

##### **app\_mutator (AppMutator):**

The app mutator that ran the mutations.

##### **applied\_migrations (list of tuple):**

The list of migrations that were ultimately marked as applied.

##### **mutations (list of BaseMutation):**

The optimized list of mutations.

##### **sql (list):**

The optimized list of SQL statements to execute.

##### **upgrade\_method (unicode):**

The resulting upgrade method for the app, after applying all mutations.

If there are no mutations to run after optimization, this will return None.

### Return type

`dict`

#### `prepare(hinted=False, **kwargs)`

Prepare state for this task.

This will determine if there are any unapplied evolutions in the app, and record that state and the SQL needed to apply the evolutions.

### Parameters

- `hinted` (`bool`, *optional*) – Whether to prepare the task for hinted evolutions.
- `**kwargs` (`dict`, *unused*) – Additional keyword arguments passed for task preparation.

#### `execute(cursor=None, sql_executor=None, sql=None, evolutions=None, create_models_now=False)`

Execute the task.

This will apply any evolutions queued up for the app.

Before the evolutions are applied for the app, the `applying_evolution` signal will be emitted. After, `applied_evolution` will be emitted.

Changed in version 2.1:

- Added `sql` and `evolutions` arguments.
- Deprecated `cursor` in favor of `sql_executor`.

### Parameters

- `cursor` (`django.db.backends.util.CursorWrapper`, *unused*) – The legacy database cursor. This is no longer used.
- `sql_executor` (`djangoevolution.utils.sql.SQLExecutor`) – The SQL executor used to run any SQL on the database.
- `sql` (`list`, *optional*) – A list of explicit SQL statements to execute.

This will override `sql` if provided.

- `evolutions` (`list` of `djangoevolution.models.Evolution`, *optional*) – A list of evolutions being applied. These will be sent in the `applying_evolution` and `applied_evolution` signals.

This will override `new_evolutions` if provided.

- `create_models_now` (`bool`, *optional*) – Whether to create models as part of this execution. Normally, this is handled in `execute_tasks()`, but this flag allows for more fine-grained control of table creation in limited circumstances (intended only by Evolver).

### Raises

`djangoevolution.errors.EvolutionExecutionError` – The evolution task failed. Details are in the error.

#### `get_evolution_content()`

Return the content for an evolution file for this task.

### Returns

The evolution content.

**Return type**  
`unicode`

**\_\_str\_\_()**

Return a string description of the task.

**Returns**  
The string description.

**Return type**  
`unicode`

## `django_evolution.evolve.purge_app_task`

Task for purging an application.

New in version 2.2: This was previously located in `django_evolution.evolve`.

### Classes

---

<code>PurgeAppTask(evolver, app_label)</code>	A task for purging an application's tables from the database.
---	---

---

**class** `django_evolution.evolve.purge_app_task.PurgeAppTask(evolver, app_label)`

Bases: `BaseEvolutionTask`

A task for purging an application's tables from the database.

#### `app_label`

The app label for the app to purge.

**Type**  
`unicode`

**\_\_init\_\_(evolver, app\_label)**

Initialize the task.

#### Parameters

- **evolver** (`Evolver`) – The evolver that will execute the task.
- **app\_label** (`unicode`) – The app label for the app to purge.

**prepare(\*\*kwargs)**

Prepare state for this task.

This will determine if the app's tables need to be deleted from the database, and prepare the SQL for doing so.

#### Parameters

`**kwargs (dict, unused)` – Keyword arguments passed for task preparation.

**execute(cursor=None, sql\_executor=None, \*\*kwargs)**

Execute the task.

This will delete any tables owned by the application.

#### Parameters

- **cursor** (`django.db.backends.util.CursorWrapper`, *unused*) – The legacy database cursor. This is no longer used.
- **sql\_executor** (`django_evolution.utils.sql.SQLExecutor`, *optional*) – The SQL executor used to run any SQL on the database.

### Raises

`django_evolution.errors.EvolutionExecutionError` – The evolution task failed.  
Details are in the error.

### `__str__()`

Return a string description of the task.

### Returns

The string description.

### Return type

`unicode`

## `django_evolution.models`

Database models for tracking project schema history.

### Classes

---

`Evolution`(`id`, `version`, `app_label`, `label`)

---

`SignatureField`([`verbose_name`, `name`, ...]) A field for loading and storing project signatures.

---

`Version`(`id`, `signature`, `when`)

---

`VersionManager`(\*args, \*\*kwargs) Manage Version models.

`class django_evolution.models.VersionManager(*args, **kwargs)`

Bases: `Manager`

Manage Version models.

This introduces a convenience function for finding the current Version model for the database.

`current_version(using=None)`

Return the Version model for the current schema.

This will find the Version with both the latest timestamp and the latest ID. It's here as a replacement for the old call to `latest()`, which only operated on the timestamp and would find the wrong entry if two had the same exact timestamp.

### Parameters

`using(unicode)` – The database alias name to use for the query. Defaults to `None`, the default database.

### Raises

`Version.DoesNotExist` – No such version exists.

### Returns

The current Version object for the database.

**Return type***Version*`__slotnames__ = []`

```
class django_evolution.models.SignatureField(verbose_name=None, name=None, primary_key=False,
                                              max_length=None, unique=False, blank=False,
                                              null=False, db_index=False, rel=None, default=<class
                                              'django.db.models.fields.NOT_PROVIDED'>,
                                              editable=True, serialize=True, unique_for_date=None,
                                              unique_for_month=None, unique_for_year=None,
                                              choices=None, help_text='', db_column=None,
                                              db_tablespace=None, auto_created=False, validators=(),
                                              error_messages=None)
```

Bases: `TextField`

A field for loading and storing project signatures.

This will handle deserializing any project signatures stored in the database, converting them into a `ProjectSignature`, and then writing a serialized version back to the database.

`description = 'Signature'`

`contribute_to_class(cls, name)`

Perform operations when added to a class.

This will listen for when an instance is constructed in order to perform some initial work.

**Parameters**

- `cls (type)` – The model class.
- `name (str)` – The name of the field.

`value_to_string(obj)`

Return a serialized string value from the field.

**Parameters**

`obj (django.db.models.Model)` – The model instance.

**Returns**

The serialized string contents.

**Return type***unicode*

`to_python(value)`

Return a `ProjectSignature` value from the field contents.

**Parameters**

`value (object)` – The current value assigned to the field. This might be serialized string content or a `ProjectSignature` instance.

**Returns**

The project signature stored in the field.

**Return type***django\_evolution.signatures.ProjectSignature***Raises**

`django.core.exceptions.ValidationError` – The field contents are of an unexpected type.

### `get_prep_value(value)`

Return a prepared Python value to work with.

This simply wraps `to_python()`.

#### Parameters

- `value (object)` – The current value assigned to the field. This might be serialized string content or a `ProjectSignature` instance.

#### Returns

The project signature stored in the field.

#### Return type

`django_evolution.signatures.ProjectSignature`

#### Raises

- `django.core.exceptions.ValidationError` – The field contents are of an unexpected type.

### `get_db_prep_value(value, connection, prepared=False)`

Return a prepared value for use in database operations.

#### Parameters

- `value (object)` – The current value assigned to the field. This might be serialized string content or a `ProjectSignature` instance.
- `connection (django.db.backends.base.BaseDatabaseWrapper)` – The database connection to operate on.
- `prepared (bool, optional)` – Whether the value is already prepared for Python.

#### Returns

The value prepared for database operations.

#### Return type

`unicode`

## `class django_evolution.models.Version(id, signature, when)`

Bases: `Model`

### `signature`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

### `when`

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

### `objects = <django_evolution.models.VersionManager object>`

### `is_hinted()`

Return whether this is a hinted version.

Hinted versions store a signature without any accompanying evolutions.

#### Returns

True if this is a hinted evolution. False if it's based on explicit evolutions.

#### Return type

`bool`

**\_\_str\_\_()**

Return str(self).

**evolutions**

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`get_next_by_when(*, field=<django.db.models.fields.DateTimeField: when>, is_next=True, **kwargs)`

`get_previous_by_when(*, field=<django.db.models.fields.DateTimeField: when>, is_next=False, **kwargs)`

**id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`class django_evolution.models.Evolution(id, version, app_label, label)`

Bases: Model

**version**

Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

Child.parent is a ForwardManyToOneDescriptor instance.

**app\_label**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**label**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**\_\_str\_\_()**

Return str(self).

**id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`objects = <django.db.models.manager.Manager object>`

**version\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

### `django_evolution.mutations`

Mutations for models, fields, and applications.

Changed in version 2.2: The classes have all been moved to nested modules. This module will provide forwarding imports, and will continue to be the primary place to import these mutations.

<code>AddField</code>	A mutation that adds a field to a model.
<code>BaseModelFieldMutation</code>	Base class for any fields that mutate a model.
<code>BaseModelMutation</code>	Base class for a mutation affecting a single model.
<code>BaseUpgradeMethodMutation</code>	Base class for a mutation that changes an app's upgrade method.
<code>BaseMutation</code>	Base class for a schema mutation.
<code>Simulation</code>	State for a database mutation simulation.
<code>ChangeField</code>	A mutation that changes attributes on a field on a model.
<code>ChangeMeta</code>	A mutation that changes meta properties on a model.
<code>DeleteApplication</code>	A mutation that deletes an application.
<code>DeleteField</code>	A mutation that deletes a field from a model.
<code>DeleteModel</code>	A mutation that deletes a model.
<code>MoveToDjangoMigrations</code>	A mutation that uses Django migrations for an app's future upgrades.
<code>RenameAppLabel</code>	A mutation that renames the app label for an application.
<code>RenameField</code>	A mutation that renames a field on a model.
<code>RenameModel</code>	A mutation that renames a model.
<code>SQLMutation</code>	A mutation that executes SQL on the database.

### `django_evolution.mutations.add_field`

Mutation that adds a field to a model.

New in version 2.2.

#### Classes

<code>AddField(model_name, field_name, field_type)</code>	A mutation that adds a field to a model.
<code>class django_evolution.mutations.add_field.AddField(model_name, field_name, field_type, initial=None, **field_attrs)</code>	
Bases: <code>BaseModelFieldMutation</code>	
A mutation that adds a field to a model.	
Changed in version 2.2: Moved into the <code>django_evolution.mutations.add_field</code> module.	
<code>simulation_failure_error = 'Cannot add the field "%(field_name)s" to model %(app_label)s.%s.(model_name)s.'</code>	
<code>__init__(model_name, field_name, field_type, initial=None, **field_attrs)</code>	
Initialize the mutation.	

#### Parameters

- `model_name` (`unicode`) – The name of the model to add the field to.

- **field\_name** (`unicode`) – The name of the new field.
- **field\_type** (`cls`) – The field class to use. This must be a subclass of `django.db.models.Field`.
- **initial** (`object`, *optional*) – The initial value for the field. This is required if non-null.
- **\*\*field\_attrs** (`dict`) – Attributes to set on the field.

**get\_hint\_params()**

Return parameters for the mutation's hinted evolution.

**Returns**

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

**Return type**

`list of unicode`

**simulate(simulation)**

Simulate the mutation.

This will alter the database schema to add the specified field.

**Parameters**

**simulation** (`Simulation`) – The state for the simulation.

**Raises**

`djangoevolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

**mutate(mutator, model)**

Schedule a field addition on the mutator.

This will instruct the mutator to add a new field on a model. It will be scheduled and later executed on the database, if not optimized out.

**Parameters**

- **mutator** (`djangoevolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- **model** (`MockModel`) – The model being mutated.

**add\_column(mutator, model)**

Add a standard column to the model.

**Parameters**

- **mutator** (`djangoevolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- **model** (`MockModel`) – The model being mutated.

**add\_m2m\_table(mutator, model)**

Add a ManyToMany column to the model and an accompanying table.

**Parameters**

- **mutator** (`djangoevolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- **model** (`MockModel`) – The model being mutated.

### `django_evolution.mutations.base`

Base support for mutations.

New in version 2.2.

#### Classes

<code>BaseModelFieldMutation(model_name, field_name)</code>	Base class for any fields that mutate a model.
<code>BaseModelMutation(model_name)</code>	Base class for a mutation affecting a single model.
<code>BaseMutation()</code>	Base class for a schema mutation.
<code>BaseUpgradeMethodMutation()</code>	Base class for a mutation that changes an app's upgrade method.
<code>Simulation(mutation, app_label, project_sig, ...)</code>	State for a database mutation simulation.

```
class django_evolution.mutations.base.Simulation(mutation, app_label, project_sig, database_state,
                                                 legacy_app_label=None, database='default')
```

Bases: `object`

State for a database mutation simulation.

This provides state and utility functions for simulating a mutation on a database signature. This is provided to `BaseMutation.simulate()` functions, given them access to all simulation state and a consistent way of failing simulations.

Changed in version 2.2: Moved into the `django_evolution.mutations.base` module.

```
__init__(mutation, app_label, project_sig, database_state, legacy_app_label=None, database='default')
```

Initialize the simulation state.

#### Parameters

- `mutation (BaseMutation)` – The mutation this simulation applies to.
- `app_label (unicode)` – The name of the application this simulation applies to.
- `project_sig (dict)` – The project signature for the simulation to look up and modify.
- `database_state (django_evolution.db.state.DatabaseState)` – The database state for the simulation to look up and modify.
- `legacy_app_label (unicode, optional)` – The legacy label of the app this simulation applies to. This is based on the module name and is used in the transitioning of pre-Django 1.7 signatures.
- `database (unicode, optional)` – The registered database name in Django to simulate operating on.

#### `get_evolver()`

Return an evolver for the database.

#### >Returns

The database evolver for this type of database.

#### Return type

`django_evolution.db.EvolutionOperationsMulti`

**get\_app\_sig()**

Return the current application signature.

**Returns**

The application signature.

**Return type**

`dict`

**Returns**

The signature for the app.

**Return type**

`django_evolution.signature.AppSignature`

**Raises**

`django_evolution.errors.SimulationFailure` – A signature could not be found for the application.

**get\_model\_sig(model\_name)**

Return the signature for a model with the given name.

**Parameters**

`model_name (unicode)` – The name of the model to fetch a signature for.

**Returns**

The signature for the model.

**Return type**

`django_evolution.signature.ModelSignature`

**Raises**

`django_evolution.errors.SimulationFailure` – A signature could not be found for the model or its parent application.

**get\_field\_sig(model\_name, field\_name)**

Return the signature for a field with the given name.

**Parameters**

- `model_name (unicode)` – The name of the model containing the field.
- `field_name (unicode)` – The name of the field to fetch a signature for.

**Returns**

The signature for the field.

**Return type**

`django_evolution.signature.FieldSignature`

**Raises**

`django_evolution.errors.SimulationFailure` – A signature could not be found for the field, its parent model, or its parent application.

**fail(error, \*\*error\_vars)**

Fail the simulation.

This will end up raising a `SimulationFailure` with an error message based on the mutation's simulation failed message an the provided message.

**Parameters**

- `error (unicode)` – The error message for this particular failure.

- **\*\*error\_vars** (`dict`) – Variables to include in the error message. These will override any defaults for the mutation’s error.

### Raises

`django_evolution.errors.SimulationFailure` – The resulting simulation failure with the given error.

```
class django_evolution.mutations.base.BaseMutation
```

Bases: `object`

Base class for a schema mutation.

These are responsible for simulating schema mutations and applying actual mutations to a database signature.

Changed in version 2.2: Moved into the `django_evolution.mutations.base` module.

```
simulation_failure_error = 'Cannot simulate the mutation.'
```

```
error_vars = {}
```

```
generate_hint()
```

Return a hinted evolution for the mutation.

This will generate a line that will be used in a hinted evolution file. This method generally should not be overridden. Instead, use `get_hint_params()`.

### Returns

A hinted evolution statement for this mutation.

### Return type

`unicode`

```
get_hint_params()
```

Return parameters for the mutation’s hinted evolution.

### Returns

A list of parameter strings to pass to the mutation’s constructor in a hinted evolution.

### Return type

`list of unicode`

```
generate_dependencies(app_label, **kwargs)
```

Return automatic dependencies for the parent evolution.

This allows a mutation to affect the order in which the parent evolution is applied, relative to other evolutions or migrations.

New in version 2.1.

### Parameters

- **app\_label** (`unicode`) – The label of the app containing this mutation.
- **\*\*kwargs** (`dict`) – Additional keyword arguments, for future use.

### Returns

A dictionary of dependencies. This may have zero or more of the following keys:

- `before_migrations`
- `after_migrations`
- `before_evolutions`
- `after_evolutions`

**Return type**`dict`**`run_simulation(**kwargs)`**

Run a simulation for a mutation.

This will prepare and execute a simulation on this mutation, constructing a *Simulation* and passing it to `simulate()`. The simulation will apply a mutation on the provided database signature, modifying it to match the state described to the mutation. This allows Django Evolution to test evolutions before they hit the database.

**Parameters**

`simulation (Simulation)` – The state for the simulation.

**Raises**

- `djangoevolution.errors.CannotSimulate` – The simulation cannot be executed for this mutation. The reason is in the exception's message.
- `djangoevolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

**`simulate(simulation)`**

Perform a simulation of a mutation.

This will attempt to perform a mutation on the database signature, modifying it to match the state described to the mutation. This allows Django Evolution to test evolutions before they hit the database.

**Parameters**

`simulation (Simulation)` – The state for the simulation.

**Raises**

- `djangoevolution.errors.CannotSimulate` – The simulation cannot be executed for this mutation. The reason is in the exception's message.
- `djangoevolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

**`mutate(mutator)`**

Schedule a database mutation on the mutator.

This will instruct the mutator to perform one or more database mutations for an app. Those will be scheduled and later executed on the database, if not optimized out.

**Parameters**

`mutator (django_evolution.mutators.AppMutator)` – The mutator to perform an operation on.

**Raises**

`djangoevolution.errors.EvolutionNotImplementedError` – The configured mutation is not supported on this type of database.

**`is_mutable(app_label, project_sig, database_state, database)`**

Return whether the mutation can be applied to the database.

This should check if the database or parts of the signature matches the attributes provided to the mutation.

**Parameters**

- `app_label (unicode)` – The label for the Django application to be mutated.
- `project_sig (dict)` – The project's schema signature.

- **database\_state** ([django\\_evolution.db.state.DatabaseState](#)) – The database’s schema signature.
- **database** ([unicode](#)) – The name of the database the operation would be performed on.

### Returns

True if the mutation can run. False if it cannot.

### Return type

[bool](#)

## `serialize_value(value)`

Serialize a value for use in a mutation statement.

This will attempt to represent the value as something Python can execute, across Python versions. The string representation of the value is used by default.

See [django\\_evolution.serialization.serialize\\_to\\_python\(\)](#) for details.

### Parameters

**value** ([object](#)) – The value to serialize.

### Returns

The serialized string.

### Return type

[unicode](#)

## `serialize_attr(attr_name, attr_value)`

Serialize an attribute for use in a mutation statement.

This will create a `name=value` string, with the value serialized using `serialize_value()`.

### Parameters

- **attr\_name** ([unicode](#)) – The attribute’s name.
- **attr\_value** ([object](#)) – The attribute’s value.

### Returns

The serialized attribute string.

### Return type

[unicode](#)

## `__hash__()`

Return a hash of this mutation.

### Returns

The mutation’s hash.

### Return type

[int](#)

## `__eq__(other)`

Return whether this mutation equals another.

Two mutations are equal if they’re of the same type and generate the same hinted evolution.

### Parameters

**other** ([BaseMutation](#)) – The mutation to compare against.

### Returns

True if the mutations are equal. False if they are not.

```
Return type
bool

__str__()
Return a hinted evolution for the mutation.

Returns
The hinted evolution.

Return type
unicode

__repr__()
Return a string representation of the mutation.

Returns
A string representation of the mutation.

Return type
unicode

class django_evolution.mutations.base.BaseUpgradeMethodMutation
Bases: BaseMutation

Base class for a mutation that changes an app's upgrade method.

New in version 2.2.

isMutable(*args, **kwargs)
Return whether the mutation can be applied to the database.

Parameters
• *args (tuple, unused) – Unused positional arguments.
• **kwargs (tuple, unused) – Unused positional arguments.

Returns
True, always.

Return type
bool

generate_dependencies(app_label, **kwargs)
Return automatic dependencies for the parent evolution.

This allows a mutation to affect the order in which the parent evolution is applied, relative to other evolutions or migrations.

This must be implemented by subclasses.

Parameters
• app_label (unicode) – The label of the app containing this mutation.
• **kwargs (dict) – Additional keyword arguments, for future use.

Returns
A dictionary of dependencies. This may have zero or more of the following keys:
• before_migrations
• after_migrations
• before_evolutions
```

- `after_evolutions`

**Return type**

`dict`

**mutate**(*mutator*)

Schedule an app mutation on the mutator.

As this mutation just modifies state on the signature, no actual database operations are performed. By default, this does nothing.

**Parameters**

`mutator` (`django_evolution.mutators.AppMutator`, *unused*) – The mutator to perform an operation on.

**class** `django_evolution.mutations.base.BaseModelMutation(model_name)`

Bases: `BaseMutation`

Base class for a mutation affecting a single model.

Changed in version 2.2: Moved into the `django_evolution.mutations.base` module.

`error_vars = {'model_name': 'model_name'}`

**\_\_init\_\_**(*model\_name*)

Initialize the mutation.

**Parameters**

`model_name` (`unicode`) – The name of the model being mutated.

**evolver**(*model*, *database\_state*, *database=None*)

**mutate**(*mutator*, *model*)

Schedule a model mutation on the mutator.

This will instruct the mutator to perform one or more database mutations for a model. Those will be scheduled and later executed on the database, if not optimized out.

**Parameters**

- `mutator` (`django_evolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- `model` (`MockModel`) – The model being mutated.

**Raises**

`django_evolution.errors.EvolutionNotImplementedError` – The configured mutation is not supported on this type of database.

**is\_mutable**(*app\_label*, *project\_sig*, *database\_state*, *database*)

Return whether the mutation can be applied to the database.

This will if the database matches that of the model.

**Parameters**

- `app_label` (`unicode`) – The label for the Django application to be mutated.
- `project_sig` (`dict`, *unused*) – The project's schema signature.
- `database_state` (`django_evolution.db.state.DatabaseState`, *unused*) – The database state.
- `database` (`unicode`) – The name of the database the operation would be performed on.

**Returns**

True if the mutation can run. False if it cannot.

**Return type**

`bool`

```
class django_evolution.mutations.base.BaseModelFieldMutation(model_name, field_name)
```

Bases: `BaseModelMutation`

Base class for any fields that mutate a model.

This is used for classes that perform any mutation on a model. Such mutations will be provided a model they can work with.

Operations added to the mutator by this field will be associated with that model. That will allow the operations backend to perform any optimizations to improve evolution time for the model.

Changed in version 2.2: Moved into the `django_evolution.mutations.base` module.

```
error_vars = {'field_name': 'field_name', 'model_name': 'model_name'}
```

```
__init__(model_name, field_name)
```

Initialize the mutation.

**Parameters**

- `model_name` (`unicode`) – The name of the model containing the field.
- `field_name` (`unicode`) – The name of the field to mutate.

## `django_evolution.mutations.change_field`

Mutation that changes attributes on a field.

New in version 2.2.

## Classes

<code>ChangeField</code> (model_name, field_name[, ...])	A mutation that changes attributes on a field on a model.
--	---

```
class django_evolution.mutations.change_field.ChangeField(model_name, field_name,
                                                               field_type=None, initial=None,
                                                               **field_attrs)
```

Bases: `BaseModelFieldMutation`

A mutation that changes attributes on a field on a model.

Changed in version 2.2:

- Moved into the `django_evolution.mutations.change_field` module.
- `field_type` can now be changed.

```
simulation_failure_error = 'Cannot change the field "%(field_name)s" on model
" %(app_label)s.%s.(model_name)s".'
```

### `__init__(model_name, field_name, field_type=None, initial=None, **field_attrs)`

Initialize the mutation.

#### Parameters

- **model\_name** (`unicode`) – The name of the model containing the field to change.
- **field\_name** (`unicode`) – The name of the field to change.
- **field\_type** (`type, optional`) – The new type of the field. This must be a subclass of `Field`.

New in version 2.2.

- **initial** (`object, optional`) – The initial value for the field. This is required if non-null.
- **\*\*field\_attrs** (`dict`) – Attributes to set on the field.

### `get_hint_params()`

Return parameters for the mutation's hinted evolution.

#### Returns

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

#### Return type

`list of unicode`

### `simulate(simulation)`

Simulate the mutation.

This will alter the database schema to change attributes for the specified field.

#### Parameters

- **simulation** (`Simulation`) – The state for the simulation.

#### Raises

`django_evolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

### `mutate(mutator, model)`

Schedule a field change on the mutator.

This will instruct the mutator to change attributes on a field on a model. It will be scheduled and later executed on the database, if not optimized out.

#### Parameters

- **mutator** (`django_evolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- **model** (`MockModel`) – The model being mutated.

## `django_evolution.mutations.change_meta`

Mutation that changes meta properties on a model.

New in version 2.2.

## Classes

<code>ChangeMeta(model_name, prop_name, new_value)</code>	A mutation that changes meta properties on a model.
---	---

`class django_evolution.mutations.change_meta.ChangeMeta(model_name, prop_name, new_value)`

Bases: `BaseModelMutation`

A mutation that changes meta properties on a model.

Changed in version 2.2: Moved into the `django_evolution.mutations.change_meta` module.

`simulation_failure_error = 'Cannot change the "%(prop_name)s" meta property on model "%(app_label)s.%s%(model_name)s".'`

`error_vars = {'model_name': 'model_name', 'prop_name': 'prop_name'}`

`__init__(model_name, prop_name, new_value)`

Initialize the mutation.

### Parameters

- `model_name` (`unicode`) – The name of the model to change meta properties on.
- `prop_name` (`unicode`) – The name of the property to change.
- `new_value` (`object`) – The new value for the property.

`get_hint_params()`

Return parameters for the mutation's hinted evolution.

### Returns

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

### Return type

`list of unicode`

`simulate(simulation)`

Simulate the mutation.

This will alter the database schema to change metadata on the specified model.

### Parameters

`simulation` (`Simulation`) – The state for the simulation.

### Raises

`django_evolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

`mutate(mutator, model)`

Schedule a model meta property change on the mutator.

This will instruct the mutator to change a meta property on a model. It will be scheduled and later executed on the database, if not optimized out.

### Parameters

- `mutator` (`django_evolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- `model` (`MockModel`) – The model being mutated.

### `django_evolution.mutations.delete_application`

Mutation that deletes an application.

New in version 2.2.

#### Classes

---

<code>DeleteApplication()</code>	A mutation that deletes an application.
----------------------------------	---

---

`class django_evolution.mutations.delete_application.DeleteApplication`

Bases: `BaseMutation`

A mutation that deletes an application.

Changed in version 2.2: Moved into the `django_evolution.mutations.delete_application` module.

`simulation_failure_error = 'Cannot delete the application "%(app_label)s".'`

`simulate(simulation)`

Simulate the mutation.

This will alter the database schema to delete the specified application.

##### Parameters

`simulation` (`Simulation`) – The state for the simulation.

##### Raises

`django_evolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

`mutate(mutator)`

Schedule an application deletion on the mutator.

This will instruct the mutator to delete an application, if it exists. It will be scheduled and later executed on the database, if not optimized out.

##### Parameters

`mutator` (`django_evolution.mutators.AppMutator`) – The mutator to perform an operation on.

`is_mutable(*args, **kwargs)`

Return whether the mutation can be applied to the database.

This will always return true. The mutation will safely handle the application no longer being around.

##### Parameters

- `*args` (`tuple, unused`) – Positional arguments passed to the function.
- `**kwargs` (`dict, unused`) – Keyword arguments passed to the function.

##### Returns

True, always.

##### Return type

`bool`

## `django_evolution.mutations.delete_field`

Mutation for deleting fields from a model.

New in version 2.2.

### Classes

<code><a href="#">DeleteField</a>(model_name, field_name)</code>	A mutation that deletes a field from a model.
<code>class django_evolution.mutations.delete_field.DeleteField(model_name, field_name)</code>	
Bases: <code>BaseModelFieldMutation</code>	
A mutation that deletes a field from a model.	
Changed in version 2.2: Moved into the <code>django_evolution.mutations.delete_field</code> module.	
<code>simulation_failure_error = 'Cannot delete the field "%(field_name)s" on model %(app_label)s.%s.(model_name)s'.</code>	
<code>get_hint_params()</code>	
Return parameters for the mutation's hinted evolution.	
<b>Returns</b>	
A list of parameter strings to pass to the mutation's constructor in a hinted evolution.	
<b>Return type</b>	
<code>list of unicode</code>	
<code>simulate(simulation)</code>	
Simulate the mutation.	
This will alter the database schema to remove the specified field, modifying meta fields ( <code>unique_together</code> ) if necessary.	
It will also check to make sure this is not a primary key and that the field exists.	
<b>Parameters</b>	
<code>simulation</code> ( <code>Simulation</code> ) – The state for the simulation.	
<b>Raises</b>	
<code><a href="#">django_evolution.errors.SimulationFailure</a></code> – The simulation failed. The reason is in the exception's message.	
<code>mutate(mutator, model)</code>	
Schedule a field deletion on the mutator.	
This will instruct the mutator to perform a deletion of a field on a model. It will be scheduled and later executed on the database, if not optimized out.	
<b>Parameters</b>	
<ul style="list-style-type: none"> <li>• <code>mutator</code> (<code>django_evolution.mutators.ModelMutator</code>) – The mutator to perform an operation on.</li> <li>• <code>model</code> (<code>MockModel</code>) – The model being mutated.</li> </ul>	

### `django_evolution.mutations.delete_model`

Mutation that deletes a model.

New in version 2.2.

#### Classes

---

<code>DeleteModel(model_name)</code>	A mutation that deletes a model.
--------------------------------------	----------------------------------

---

`class django_evolution.mutations.delete_model.DeleteModel(model_name)`

Bases: `BaseModelMutation`

A mutation that deletes a model.

Changed in version 2.2: Moved into the `django_evolution.mutations.delete_model` module.

`simulation_failure_error = 'Cannot delete the model "%(app_label)s.%s%(model_name)s".'`

`get_hint_params()`

Return parameters for the mutation's hinted evolution.

##### Returns

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

##### Return type

`list of unicode`

`simulate(simulation)`

Simulate the mutation.

This will alter the database schema to delete the specified model.

##### Parameters

`simulation` (`Simulation`) – The state for the simulation.

##### Raises

`django_evolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

`mutate(mutator, model)`

Schedule a model deletion on the mutator.

This will instruct the mutator to delete a model. It will be scheduled and later executed on the database, if not optimized out.

##### Parameters

- `mutator` (`django_evolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- `model` (`MockModel`) – The model being mutated.

## `django_evolution.mutations.move_to_django_migrations`

Mutation that moves an app to Django migrations.

New in version 2.2.

### Classes

<code>MoveToDjangoMigrations([mark_applied])</code>	A mutation that uses Django migrations for an app's future upgrades.
---	--

`class django_evolution.mutations.move_to_django_migrations.MoveToDjangoMigrations(mark_applied=['0001_initial'])`

Bases: `BaseUpgradeMethodMutation`

A mutation that uses Django migrations for an app's future upgrades.

This directs this app to evolve only up until this mutation, and to then hand any future schema changes over to Django's migrations.

Once this mutation is used, no further mutations can be added for the app.

Changed in version 2.2: Moved into the `django_evolution.mutations.move_to_django_migrations` module.

`__init__(mark_applied=['0001_initial'])`

Initialize the mutation.

#### Parameters

`mark_applied (unicode, optional)` – The list of migrations to mark as applied. Each of these should have been covered by the initial table or subsequent evolutions. By default, this covers the `0001_initial` migration.

`generate_dependencies(app_label, **kwargs)`

Return automatic dependencies for the parent evolution.

This will generate a dependency forcing this evolution to apply before the migrations that are marked as applied, ensuring that subsequent migrations are applied in the correct order.

New in version 2.1.

#### Parameters

- `app_label (unicode)` – The label of the app containing this mutation.
- `**kwargs (dict)` – Additional keyword arguments, for future use.

#### Returns

A dictionary of dependencies. This may have zero or more of the following keys:

- `before_migrations`
- `after_migrations`
- `before_evolutions`
- `after_evolutions`

#### Return type

`dict`

### `simulate(simulation)`

Simulate the mutation.

This will alter the app's signature to mark it as being handled by Django migrations.

#### Parameters

`simulation (Simulation)` – The simulation being performed.

## [django\\_evolution.mutations.rename\\_app\\_label](#)

Mutation that renames the app label for an application.

New in version 2.2.

## Classes

---

`RenameAppLabel(old_app_label, new_app_label)` A mutation that renames the app label for an application.

---

`class django_evolution.mutations.rename_app_label.RenameAppLabel(old_app_label, new_app_label, legacy_app_label=None, model_names=None)`

Bases: `BaseMutation`

A mutation that renames the app label for an application.

Changed in version 2.2: Moved into the `django_evolution.mutations.rename_app_label` module.

`__init__(old_app_label, new_app_label, legacy_app_label=None, model_names=None)`

`get_hint_params()`

Return parameters for the mutation's hinted evolution.

#### Returns

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

#### Return type

`list of unicode`

`is Mutable(app_label, project_sig, database_state, database)`

Return whether the mutation can be applied to the database.

#### Parameters

- `app_label (unicode)` – The label for the Django application to be mutated.
- `project_sig (dict, unused)` – The project's schema signature.
- `database_state (django_evolution.db.state.DatabaseState, unused)` – The database state.
- `database (unicode)` – The name of the database the operation would be performed on.

#### Returns

True if the mutation can run. False if it cannot.

#### Return type

`bool`

**simulate(simulation)**

Simulate the mutation.

This will alter the signature to make any changes needed for the application's evolution storage.

**mutate(mutator)**

Schedule an app mutation on the mutator.

This will inform the mutator of the new app label, for use in any future operations.

**Parameters**

**mutator** (`django_evolution.mutators.AppMutator`) – The mutator to perform an operation on.

**`django_evolution.mutations.rename_field`**

Mutation that renames a field on a model.

New in version 2.2.

**Classes**

---

<code>RenameField(model_name, old_field_name, ...)</code>	A mutation that renames a field on a model.
---	---

---

**class django\_evolution.mutations.rename\_field.RenameField(model\_name, old\_field\_name, new\_field\_name, db\_column=None, db\_table=None)**

Bases: `BaseModelFieldMutation`

A mutation that renames a field on a model.

Changed in version 2.2: Moved into the `django_evolution.mutations.rename_field` module.

`simulation_failure_error = 'Cannot rename the field "%(field_name)s" on model "%(app_label)s.%s%(model_name)s".'`

**`__init__(model_name, old_field_name, new_field_name, db_column=None, db_table=None)`**

Initialize the mutation.

**Parameters**

- **model\_name** (`unicode`) – The name of the model to add the field to.
- **old\_field\_name** (`unicode`) – The old (existing) name of the field.
- **new\_field\_name** (`unicode`) – The new name for the field.
- **db\_column** (`unicode, optional`) – The explicit column name to set for the field.
- **db\_table** (`object, optional`) – The explicit table name to use, if specifying a `ManyToManyField`.

**`get_hint_params()`**

Return parameters for the mutation's hinted evolution.

**Returns**

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

### Return type

`list of unicode`

### `simulate(simulation)`

Simulate the mutation.

This will alter the database schema to rename the specified field.

#### Parameters

`simulation` (`Simulation`) – The state for the simulation.

#### Raises

`django_evolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

### `mutate(mutator, model)`

Schedule a field rename on the mutator.

This will instruct the mutator to rename a field on a model. It will be scheduled and later executed on the database, if not optimized out.

#### Parameters

- `mutator` (`django_evolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- `model` (`MockModel`) – The model being mutated.

## `django_evolution.mutations.rename_model`

Mutation that renames a model.

New in version 2.2.

### Classes

---

`RenameModel(old_model_name, new_model_name)` A mutation that renames a model.  
...)

---

`class django_evolution.mutations.rename_model.RenameModel(old_model_name, new_model_name, db_table)`

Bases: `BaseModelMutation`

A mutation that renames a model.

Changed in version 2.2: Moved into the `django_evolution.mutations.rename_model` module.

`simulation_failure_error = 'Cannot rename the model "%(app_label)s.%s.(model_name)s".'`

`__init__(old_model_name, new_model_name, db_table)`

Initialize the mutation.

#### Parameters

- `old_model_name` (`unicode`) – The old (existing) name of the model to rename.
- `new_model_name` (`unicode`) – The new name for the model.
- `db_table` (`unicode`) – The table name in the database for this model.

**`get_hint_params()`**

Return parameters for the mutation's hinted evolution.

**Returns**

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

**Return type**

`list of unicode`

**`simulate(simulation)`**

Simulate the mutation.

This will alter the database schema to rename the specified model.

**Parameters**

`simulation` (`Simulation`) – The state for the simulation.

**Raises**

`django_evolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message.

**`mutate(mutator, model)`**

Schedule a model rename on the mutator.

This will instruct the mutator to rename a model. It will be scheduled and later executed on the database, if not optimized out.

**Parameters**

- `mutator` (`django_evolution.mutators.ModelMutator`) – The mutator to perform an operation on.
- `model` (`MockModel`) – The model being mutated.

**`django_evolution.mutations.sql_mutation`**

Mutation for executing SQL statements.

New in version 2.2.

**Classes****`SQLMutation(tag, sql[, update_func])`**

A mutation that executes SQL on the database.

**`class django_evolution.mutations.sql_mutation.SQLMutation(tag, sql, update_func=None)`**

Bases: `BaseMutation`

A mutation that executes SQL on the database.

Unlike most mutations, this one is largely database-dependent. It allows arbitrary SQL to be executed. It's recommended that the execution does not modify the schema of a table (unless it's highly database-specific with no counterpart in Django Evolution), but rather is limited to things like populating data.

SQL statements cannot be optimized. Any scheduled database operations prior to the SQL statement will be executed without any further optimization. This can lead to longer database evolution times.

Changed in version 2.2: Moved into the `django_evolution.mutations.sql_mutation` module.

### `__init__(tag, sql, update_func=None)`

Initialize the mutation.

#### Parameters

- `tag (unicode)` – A unique tag identifying this SQL operation.
- `sql (unicode)` – The SQL to execute.
- `update_func (callable, optional)` – A function to call to simulate updating the database signature. This is required for `simulate()` to work.

### `get_hint_params()`

Return parameters for the mutation's hinted evolution.

#### Returns

A list of parameter strings to pass to the mutation's constructor in a hinted evolution.

#### Return type

`list of unicode`

### `simulate(simulation)`

Simulate a mutation for an application.

This will run the `update_func` provided when instantiating the mutation, passing it `app_label` and `project_sig`. It should then modify the signature to match what the SQL statement would do.

#### Parameters

`simulation (Simulation)` – The state for the simulation.

#### Raises

- `django_evolution.errors.CannotSimulate` – `update_func` was not provided or was not a function.
- `django_evolution.errors.SimulationFailure` – The simulation failed. The reason is in the exception's message. This would be run by `update_func`.

### `mutate(mutator)`

Schedule a database mutation on the mutator.

This will instruct the mutator to execute the SQL for an app.

#### Parameters

`mutator (django_evolution.mutators.AppMutator)` – The mutator to perform an operation on.

#### Raises

`django_evolution.errors.EvolutionNotImplementedError` – The configured mutation is not supported on this type of database.

### `is_mutable(*args, **kwargs)`

Return whether the mutation can be applied to the database.

#### Parameters

- `*args (tuple, unused)` – Unused positional arguments.
- `**kwargs (tuple, unused)` – Unused positional arguments.

#### Returns

True, always.

#### Return type

`bool`

## `django_evolution.serialization`

Serialization and deserialization.

These classes are responsible for converting objects/values to signature data or to Python code (for evolution hints), and for converting signature data back to objects.

The classes in this file are considered private API. The only public API is:

- `deserialize_from_python()`
- `serialize_to_signature()`
- `serialize_to_python()`

New in version 2.2.

### Functions

<code>deserialize_from_signature(payload)</code>	Deserialize a value from the signature.
<code>serialize_to_python(value)</code>	Serialize a value to a Python code string.
<code>serialize_to_signature(value)</code>	Serialize a value to the signature.

### Classes

<code>BaseIterableSerialization()</code>	Base class for iterable types.
<code>BaseSerialization()</code>	Base class for serialization.
<code>ClassSerialization()</code>	Base class for serialization for classes.
<code>CombinedExpressionSerialization()</code>	Base class for serialization for CombinedExpression objects.
<code>DeconstructedSerialization()</code>	Base class for serialization for objects supporting deconstruction.
<code>DictSerialization()</code>	Base class for serialization for dictionaries.
<code>EnumSerialization()</code>	Serialization for enums.
<code>ListSerialization()</code>	Base class for serialization for lists.
<code>PlaceholderSerialization()</code>	Base class for serialization for a placeholder object.
<code>PrimitiveSerialization()</code>	Base class for serialization for Python primitives.
<code>QSerialization()</code>	Base class for serialization for Q objects.
<code>SetSerialization()</code>	Base class for serialization for sets.
<code>StringSerialization()</code>	Base class for serialization for strings.
<code>TupleSerialization()</code>	Base class for serialization for tuples.

#### `class django_evolution.serialization.BaseSerialization`

Bases: `object`

Base class for serialization.

Subclasses should override the methods within this class to provide serialization and deserialization logic specific to one or more types.

New in version 2.2.

**classmethod serialize\_to\_signature(value)**

Serialize a value to JSON-compatible signature data.

**Parameters**

**value** (`object` or `type`) – The value to serialize.

**Returns**

The resulting signature data.

**Return type**

`object`

**classmethod serialize\_to\_python(value)**

Serialize a value to a Python code string.

**Parameters**

**value** (`object` or `type`) – The value to serialize.

**Returns**

The resulting Python code.

**Return type**

`unicode`

**classmethod deserialize\_from\_signature(payload)**

Deserialize signature data to a value.

**Parameters**

**payload** (`object`) – The payload to deserialize.

**Returns**

The resulting value.

**Return type**

`object` or `type`

**classmethod deserialize\_from\_deconstructed(type\_cls, args, kwargs)**

Deserialize an object from deconstructed object information.

**Parameters**

- **type\_cls** (`type`) – The type of object to construct.
- **args** (`tuple`) – The positional arguments passed to the constructor.
- **kwargs** (`dict`) – The keyword arguments passed to the constructor.

**Returns**

The resulting object.

**Return type**

`object`

**class django\_evolution.serialization.BaseIterableSerialization**

Bases: `BaseSerialization`

Base class for iterable types.

This will handle the signature-related serialization/deserialization automatically, based on `iterable_type`.

New in version 2.2.

```
item_type = None
```

The type used to store items.

**Type**

```
type
```

```
classmethod serialize_to_signature(value)
```

Serialize a value to JSON-compatible signature data.

**Parameters**

**value** (`object` or `type`) – The value to serialize.

**Returns**

The resulting signature data.

**Return type**

```
object
```

```
classmethod deserialize_from_signature(payload)
```

Deserialize signature data to a value.

**Parameters**

**payload** (`object`) – The payload to deserialize.

**Returns**

The resulting value.

**Return type**

```
object or type
```

```
class django_evolution.serialization.PrimitiveSerialization
```

Bases: `BaseSerialization`

Base class for serialization for Python primitives.

This will wrap simple values, deep-copying them when storing as signature data, returning a `repr()` result when converting to Python code, and using the value as-is when deserializing.

New in version 2.2.

```
classmethod serialize_to_signature(value)
```

Serialize a value to JSON-compatible signature data.

**Parameters**

**value** (`object`) – The value to serialize.

**Returns**

A deep copy of the provided value.

**Return type**

```
object
```

```
classmethod serialize_to_python(value)
```

Serialize a value to a Python code string.

**Parameters**

**value** (`object` or `type`) – The value to serialize.

**Returns**

The resulting Python code.

**Return type**

```
unicode
```

```
classmethod deserialize_from_signature(payload)
```

Deserialize signature data to a value.

This will just return the value as-is.

**Parameters**

`payload (object)` – The payload to deserialize.

**Returns**

The resulting value.

**Return type**

`object or type`

```
class django_evolution.serialization.ClassSerialization
```

Bases: `BaseSerialization`

Base class for serialization for classes.

This is able to serialize a class name to Python. It cannot be used for signature data.

New in version 2.2.

```
classmethod serialize_to_python(value)
```

Serialize a value to a Python code string.

**Parameters**

`value (object or type)` – The value to serialize.

**Returns**

The resulting Python code.

**Return type**

`unicode`

```
class django_evolution.serialization.DictSerialization
```

Bases: `BaseSerialization`

Base class for serialization for dictionaries.

This will be used for plain `dict` instances and for `collections.OrderedDict`.

New in version 2.2.

```
classmethod serialize_to_signature(value)
```

Serialize a dictionary to JSON-compatible signature data.

**Parameters**

`value (dict)` – The dictionary to serialize.

**Returns**

The resulting dictionary.

**Return type**

`dict`

```
classmethod serialize_to_python(value)
```

Serialize a dictionary to a Python code string.

**Parameters**

`value (dict)` – The dictionary to serialize.

**Returns**

The resulting Python code.

```
Return type
unicode

classmethod deserialize_from_signature(payload)
    Deserialize dictionary signature data to a value.

    Parameters
        payload (dict) – The payload to deserialize.

    Returns
        The resulting value.

    Return type
        dict

class django_evolution.serialization.EnumSerialization
    Bases: BaseSerialization

    Serialization for enums.

    New in version 2.2.

    classmethod serialize_to_signature(value)
        Serialize a value to JSON-compatible signature data.

        Parameters
            value (object) – The value to serialize.

        Returns
            A deep copy of the provided value.

        Return type
            object

    classmethod serialize_to_python(value)
        Serialize an enum value to a Python code string.

        Parameters
            value (enum.Enum) – The enum value to serialize.

        Returns
            The resulting Python code.

        Return type
            unicode

    classmethod deserialize_from_signature(payload)
        Deserialize signature data to a value.

        This will just return the value as-is.

        Parameters
            payload (object) – The payload to deserialize.

        Returns
            The resulting value.

        Return type
            object or type
```

```
class django_evolution.serialization.ListSerialization
Bases: BaseIterableSerialization
```

Base class for serialization for lists.

New in version 2.2.

**item\_type**

alias of `list`

```
classmethod serialize_to_python(value)
```

Serialize a list to a Python code string.

**Parameters**

`value (list)` – The list to serialize.

**Returns**

The resulting Python code.

**Return type**

`unicode`

```
class django_evolution.serialization.TupleSerialization
```

Bases: `BaseIterableSerialization`

Base class for serialization for tuples.

New in version 2.2.

**item\_type**

alias of `tuple`

```
classmethod serialize_to_python(value)
```

Serialize a tuple to a Python code string.

**Parameters**

`value (tuple)` – The tuple to serialize.

**Returns**

The resulting Python code.

**Return type**

`unicode`

```
class django_evolution.serialization.SetSerialization
```

Bases: `BaseIterableSerialization`

Base class for serialization for sets.

New in version 2.2.

**item\_type**

alias of `set`

```
classmethod serialize_to_python(value)
```

Serialize a set to a Python code string.

**Parameters**

`value (set)` – The set to serialize.

**Returns**

The resulting Python code.

**Return type**  
`unicode`

**class** `django_evolution.serialization.StringSerialization`

Bases: *PrimitiveSerialization*

Base class for serialization for strings.

This will encode to a string, and ensure the results are consistent across Python 2 and 3.

New in version 2.2.

**classmethod** `serialize_to_signature(value)`

Serialize a string to JSON-compatible string.

**Parameters**  
`value` (`bytes` or `unicode`) – The string to serialize. If a byte string, it's expected to contain UTF-8 data.

**Returns**  
The resulting string.

**Return type**  
`unicode`

**classmethod** `serialize_to_python(value)`

Serialize a string to a Python code string.

**Parameters**  
`value` (`bytes` or `unicode`) – The string to serialize. If a byte string, it's expected to contain UTF-8 data.

**Returns**  
The resulting Python code.

**Return type**  
`unicode`

**class** `django_evolution.serialization.DeconstructedSerialization`

Bases: *BaseSerialization*

Base class for serialization for objects supporting deconstruction.

This is used for Django objects that support a `deconstruct()` method. It will convert to/from deconstructed signature data, and provide a suitable representation in Python.

New in version 2.2.

**classmethod** `serialize_to_signature(value)`

Serialize a value to JSON-compatible signature data.

This will deconstruct the object and return a dictionary containing the deconstructed information and a flag noting that it must be reconstructed.

**Parameters**  
`value` (`object` or `type`) – The value to serialize.

**Returns**  
The resulting signature data.

**Return type**  
`object`

### `classmethod serialize_to_python(value)`

Serialize an object to a Python code string.

This will generate code that constructs an instance of the object.

#### Parameters

`value (object)` – The object to serialize.

#### Returns

The resulting Python code.

#### Return type

`unicode`

### `classmethod deserialize_from_signature(payload)`

Deserialize deconstructed dictionary signature data to an object.

This will attempt to re-construct an object from the deconstructed signature data. This may fail if there is any issue looking up or instantiating the object.

#### Parameters

`payload (dict)` – The payload to deserialize.

#### Returns

The resulting value.

#### Return type

`dict`

#### Raises

- `Exception` – An unexpected error occurred when instantiating the object.
- `ImportError` – The class specified in the signature data could not be imported.

## `class django_evolution.serialization.PlaceholderSerialization`

Bases: `BaseSerialization`

Base class for serialization for a placeholder object.

New in version 2.2.

### `classmethod serialize_to_python(value)`

Serialize a placeholder object to a Python code string.

#### Parameters

`value (django_evolution.placeholders.BasePlaceholder)` – The object to serialize.

#### Returns

The resulting Python code.

#### Return type

`unicode`

## `class django_evolution.serialization.CombinedExpressionSerialization`

Bases: `DeconstructedSerialization`

Base class for serialization for CombinedExpression objects.

This ensures a consistent representation of `django.db.models.CombinedExpression` objects across all supported versions of Django.

Note that while this can technically be used in version of Django prior to 2.0, many of the objects nested within won't be supported. In practice, database features really start to make use of this in a way that impacts serialization code in Django 2.0 and higher.

New in version 2.2.

**classmethod serialize\_to\_python(value)**

Serialize a CombinedExpression object to a Python code string.

**Parameters**

**value** (`object`) – The object to serialize.

**Returns**

The resulting Python code.

**Return type**

`unicode`

**class django\_evolution.serialization.QSerialization**

Bases: `DeconstructedSerialization`

Base class for serialization for Q objects.

This ensures a consistent representation of `django.db.models.Q` objects across all supported versions of Django.

Django 1.7 through 3.1 encode the data in a different form than 3.2+. This ensures serialized data in a form closer to 3.2+'s version, while providing compatibility with older versions.

New in version 2.2.

**child\_separators = {'AND': ' & ', 'OR': ' | '}**

**classmethod serialize\_to\_signature(q)**

Serialize a Q object to JSON-compatible signature data.

**Parameters**

**value** (`object` or `type`) – The value to serialize.

**Returns**

The resulting signature data.

**Return type**

`object`

**classmethod serialize\_to\_python(value)**

Serialize a Q object to a Python code string.

This will generate code that constructs an instance of the object, handling negation, AND/OR connections, and children.

**Parameters**

**value** (`object`) – The object to serialize.

**Returns**

The resulting Python code.

**Return type**

`unicode`

**classmethod deserialize\_from\_deconstructed(type\_cls, args, kwargs)**

Deserialize an object from deconstructed object information.

**Parameters**

- **type\_cls** (`type`) – The type of object to construct.
- **args** (`tuple`) – The positional arguments passed to the constructor.
- **kwargs** (`dict`) – The keyword arguments passed to the constructor.

**Returns**

The resulting object.

**Return type**

`object`

`django_evolution.serialization.serialize_to_signature(value)`

Serialize a value to the signature.

New in version 2.2.

**Parameters**

`value` (`object` or `type`) – The value to serialize.

**Returns**

The resulting JSON-serializable data.

**Return type**

`object`

`django_evolution.serialization.serialize_to_python(value)`

Serialize a value to a Python code string.

New in version 2.2.

**Parameters**

`value` (`object` or `type`) – The value to serialize.

**Returns**

The resulting Python code string.

**Return type**

`unicode`

`django_evolution.serialization.deserialize_from_signature(payload)`

Deserialize a value from the signature.

New in version 2.2.

**Parameters**

`payload` (`object`) – The payload to deserialize.

**Returns**

The resulting deserialized value.

**Return type**

`object` or `type`

**Raises**

`Exception` – An unexpected error occurred when deserializing. This is specific to the type of deserializer.

## `django_evolution.signals`

Signals for monitoring the evolution process.

### Module Attributes

<code>evolving</code>	Emitted when an Evolver begins evolving.
<code>evolved</code>	Emitted when an Evolver finishes evolving.
<code>evolving_failed</code>	Emitted when an Evolver fails evolving.
<code>applying_evolution</code>	Emitted when an evolution is about to be applied to an app.
<code>applied_evolution</code>	Emitted when an evolution has been applied to an app.
<code>applying_migration</code>	Emitted when a migration is about to be applied to an app.
<code>applied_migration</code>	Emitted when a migration has been applied to an app.
<code>creating_models</code>	Emitted when creating new models for an app outside of a migration.
<code>created_models</code>	Emitted when finished creating new models for an app outside of a migration.

`django_evolution.signals.evolving = <django.dispatch.dispatcher.Signal object>`

Emitted when an Evolver begins evolving.

`django_evolution.signals.evolved = <django.dispatch.dispatcher.Signal object>`

Emitted when an Evolver finishes evolving.

`django_evolution.signals.evolving_failed = <django.dispatch.dispatcher.Signal object>`

Emitted when an Evolver fails evolving.

#### Parameters

`exception` (`Exception`) – The exception raised when evolution failed.

`django_evolution.signals.applying_evolution = <django.dispatch.dispatcher.Signal object>`

Emitted when an evolution is about to be applied to an app.

Changed in version 2.1: Added the `evolutions` argument.

#### Parameters

- `app_label` (`unicode`) – The label of the application being applied.
- `task` (`django_evolution.evolve.EvolveAppTask`) – The task evolving the app.
- `evolutions` (`list` of `django_evolution.models.Evolution`) – The list of evolutions that will be applied.

`django_evolution.signals.applied_evolution = <django.dispatch.dispatcher.Signal object>`

Emitted when an evolution has been applied to an app.

Changed in version 2.1: Added the `evolutions` argument.

#### Parameters

- `app_label` (`unicode`) – The label of the application being applied.
- `task` (`django_evolution.evolve.EvolveAppTask`) – The task that evolved the app.

- **evolutions** (*list of django\_evolution.models.Evolution*) – The list of evolutions that were applied.

`django_evolution.signals.applying_migration = <django.dispatch.dispatcher.Signal object>`

Emitted when a migration is about to be applied to an app.

#### Parameters

- **migration** (*django.db.migrations.migration.Migration*) – The migration that's being applied.

`django_evolution.signals.applied_migration = <django.dispatch.dispatcher.Signal object>`

Emitted when a migration has been applied to an app.

#### Parameters

- **migration** (*django.db.migrations.migration.Migration*) – The migration that was applied.

`django_evolution.signals.creating_models = <django.dispatch.dispatcher.Signal object>`

Emitted when creating new models for an app outside of a migration.

---

**Note:** There's no guarantee that a *created\_models* will be emitted in-between two *creating\_models*.

---

#### Parameters

- **app\_label** (*unicode*) – The app label for the models being created.
- **model\_names** (*list of unicode*) – The list of models being created.

`django_evolution.signals.created_models = <django.dispatch.dispatcher.Signal object>`

Emitted when finished creating new models for an app outside of a migration.

---

**Note:** There's no guarantee that a *creating\_models* will be emitted in-between two *created\_models*.

---

#### Parameters

- **migration** (*django.db.migrations.migration.Migration*) – The migration that was applied.
- **model\_names** (*list of unicode*) – The list of models that were created.

## `django_evolution.signature`

Classes for working with stored evolution state signatures.

These provide a way to work with the state of Django apps and their models in an abstract way, and to deserialize from or serialize to a string. Signatures can also be diffed, showing the changes between an older and a newer version in order to help see how the current database's signature differs from an older stored version.

Serialized versions of signatures are versioned, and the signature classes handle loading and saving as any version. However, state may be lost when downgrading a signature.

The following versions are currently supported:

#### Version 1:

The original version of the signature, used up until Django Evolution 1.0. This is in the form of:

```
{
    '__version__': 1,
    '<legacy_app_label>': {
        '<model_name>': {
            'meta': {
                'db_table': '<table name>',
                'db_tablespace': '<tablespace>',
                'index_together': [
                    ('<colname>', ...),
                    ...
                ],
                'indexes': [
                    {
                        'name': '<name>',
                        'fields': ['<colname>', ...],
                    },
                    ...
                ],
                'pk_column': '<colname>',
                'unique_together': [
                    ('<colname>', ...),
                    ...
                ],
                '__unique_together_applied': True|False,
            },
            'fields': {
                'field_type': <class>,
                'related_model': '<app_label>.<class_name>',
                '<field_attr>': <value>,
                ...
            },
            ...
        },
        ...
    },
    ...
}
```

**Version 2:**

Introduced in Django Evolution 2.0. This differs from version 1 in that it's deeper, with explicit namespaces for apps, models, and field attributes that can exist alongside metadata keys. This is in the form of:

```
{
    '__version__': 2,
    'apps': {
        '<app_label>': {
            'legacy_app_label': '<legacy app_label>',
            'upgrade_method': 'migrations' | 'evolutions' | None,
            'applied_migrations': ['<migration name>', ...],
            'models': {
                '<model_name>': {
                    'meta': {
                        'constraints': [
                            {

```

(continues on next page)

(continued from previous page)

```
        'name': '<name>',
        'type': '<class_path>',
        'attrs': {
            '<attr_name>': <value>,
        },
    },
    ...
],
'db_table': '<table name>',
'db_tablespace': '<tablespace>',
'index_together': [
    ('<colname>', ...),
    ...
],
'indexes': [
    {
        'name': '<name>',
        'fields': ['<colname>', ...],
        'expressions': [
            {<deconstructed>},
            ...
        ],
        'attrs': {
            'condition': {<deconstructed>},
            'db_tablespace': '<string>',
            'include': ['<name>', ...],
            'opclasses': ['<name>', ...],
        },
    },
    ...
],
'pk_column': '<colname>',
'unique_together': [
    ('<colname>', ...),
    ...
],
'__unique_together_applied': True|False,
},
'fields': {
    'type': '<class_path>',
    'related_model': '<app_label>.<class_name>',
    'attrs': {
        '<field_attr_name>': <value>,
        ...
    },
},
...
},
...
},
...
},
...
}
,
```

(continues on next page)

(continued from previous page)

}

## Module Attributes

<code>LATEST_SIGNATURE_VERSION</code>	The latest signature version.
---------------------------------------	-------------------------------

## Functions

<code>validate_sig_version(sig_version)</code>	Validate that a signature version is supported.
--	---

## Classes

<code>AppSignature(app_id[, legacy_app_label, ...])</code>	Signature information for an application.
<code>BaseSignature()</code>	Base class for a signature.
<code>ConstraintSignature(name, constraint_type[, ...])</code>	Signature information for a explicit constraint.
<code>FieldSignature(field_name, field_type[, ...])</code>	Signature information for a field.
<code>IndexSignature(fields[, name, expressions, ...])</code>	Signature information for an explicit index.
<code>ModelSignature(model_name, table_name[, ...])</code>	Signature information for a model.
<code>ProjectSignature()</code>	Signature information for a project.

`django_evolution.signature.LATEST_SIGNATURE_VERSION = 2`

The latest signature version.

`class django_evolution.signature.BaseSignature`

Bases: `object`

Base class for a signature.

`classmethod deserialize(sig_dict, sig_version, database='default')`

Deserialize the signature.

### Parameters

- `sig_dict (dict)` – The dictionary containing signature data.
- `sig_version (int)` – The stored signature version.
- `database (unicode, optional)` – The name of the database.

### Returns

The resulting signature class.

### Return type

`BaseSignature`

### Raises

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

### `diff(old_sig)`

Diff against an older signature.

The resulting data is dependent on the type of signature.

#### **Parameters**

`old_sig` (`BaseSignature`) – The old signature to diff against.

#### **Returns**

The resulting diffed data.

#### **Return type**

`object`

### `clone()`

Clone the signature.

#### **Returns**

The cloned signature.

#### **Return type**

`BaseSignature`

### `serialize(sig_version=2)`

Serialize data to a signature dictionary.

#### **Parameters**

`sig_version` (`int, optional`) – The signature version to serialize as. This always defaults to the latest.

#### **Returns**

The serialized data.

#### **Return type**

`dict`

#### **Raises**

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

### `__eq__(other)`

Return whether two signatures are equal.

#### **Parameters**

`other` (`BaseSignature`) – The other signature.

#### **Returns**

True if the project signatures are equal. False if they are not.

#### **Return type**

`bool`

### `__ne__(other)`

Return whether two signatures are not equal.

#### **Parameters**

`other` (`BaseSignature`) – The other signature.

#### **Returns**

True if the project signatures are not equal. False if they are equal.

#### **Return type**

`bool`

**`__repr__()`**

Return a string representation of the signature.

**Returns**

A string representation of the signature.

**Return type**

`unicode`

**`__hash__ = None`****`class django_evolution.signature.ProjectSignature`**

Bases: `BaseSignature`

Signature information for a project.

Projects are the top-level signature deserialized from and serialized to a `Version` model. They contain a signature version and information on all the applications tracked for the project.

**`classmethod from_database(database)`**

Create a project signature from the database.

This will look up all the applications registered in Django, turning each of them into a `AppSignature` stored in this project signature.

**Parameters**

`database` (`unicode`) – The name of the database.

**Returns**

The project signature based on the current application and database state.

**Return type**

`ProjectSignature`

**`classmethod deserialize(project_sig_dict, database='default')`**

Deserialize a serialized project signature.

**Parameters**

- `project_sig_dict` (`dict`) – The dictionary containing project signature data.
- `database` (`unicode`, *optional*) – The name of the database.

**Returns**

The resulting signature instance.

**Return type**

`ProjectSignature`

**Raises**

`django_evolution.errors.InvalidSignatureVersion` – The signature version found in the dictionary is unsupported.

**`__init__()`**

Initialize the signature.

**`property app_sigs`**

The application signatures in the project signature.

**`add_app(app, database)`**

Add an application to the project signature.

This will construct an `AppSignature` and add it to the project signature.

### Parameters

- **app** (`module`) – The application module to create the signature from.
- **database** (`unicode`) – The database name.

### `add_app_sig(app_sig)`

Add an application signature to the project signature.

### Parameters

`app_sig` (`AppSignature`) – The application signature to add.

### `remove_app_sig(app_id)`

Remove an application signature from the project signature.

### Parameters

`app_id` (`unicode`) – The ID of the application signature to remove.

### Raises

`djangoevolution.errors.MissingSignatureError` – The application ID does not represent a known application signature.

### `get_app_sig(app_id, required=False)`

Return an application signature with the given ID.

### Parameters

- **app\_id** (`unicode`) – The ID of the application signature. This may be a modern app label, or a legacy app label.
- **required** (`bool, optional`) – Whether the app signature must be present. If `True` and the signature is missing, this will raise an exception.

### Returns

The application signature, if found. If no application signature matches the ID, `None` will be returned.

### Return type

`AppSignature`

### Raises

`djangoevolution.errors.MissingSignatureError` – The application signature was not found, and `required` was `True`.

### `diff(old_project_sig)`

Diff against an older project signature.

This will return a dictionary of changes between two project signatures.

### Parameters

`old_project_sig` (`ProjectSignature`) – The old project signature to diff against.

### Returns

A dictionary in the following form:

```
{  
    'changed': {  
        <app ID>: <AppSignature diff>,  
        ...  
    },  
    'deleted': [  
        ...  
    ]  
}
```

(continues on next page)

(continued from previous page)

```
<app ID>: [
    <model name>,
    ...
],
...
}
```

Any key lacking a value will be omitted from the diff.

#### Return type

`collections.OrderedDict`

#### Raises

`TypeError` – The old signature provided was not a `ProjectSignature`.

#### `clone()`

Clone the signature.

#### Returns

The cloned signature.

#### Return type

`ProjectSignature`

#### `serialize(sig_version=2)`

Serialize project data to a signature dictionary.

#### Parameters

`sig_version` (`int`, *optional*) – The signature version to serialize as. This always defaults to the latest.

#### Returns

The serialized data.

#### Return type

`dict`

#### Raises

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

#### `__eq__(other)`

Return whether two project signatures are equal.

#### Parameters

`other` (`ProjectSignature`) – The other project signature.

#### Returns

True if the project signatures are equal. False if they are not.

#### Return type

`bool`

#### `__repr__()`

Return a string representation of the signature.

#### Returns

A string representation of the signature.

**Return type**

`unicode`

`__hash__ = None`

```
class django_evolution.signature.AppSignature(app_id, legacy_app_label=None,
                                             upgrade_method=None, applied_migrations=None)
```

Bases: `BaseSignature`

Signature information for an application.

Application signatures store information on a Django application and all models registered under that application.

`classmethod from_app(app, database)`

Create an application signature from an application.

This will store data on the application and create a `ModelSignature` for each of the application's models.

**Parameters**

- `app (module)` – The application module to create the signature from.
- `database (unicode)` – The name of the database.

**Returns**

The application signature based on the application.

**Return type**

`AppSignature`

```
classmethod deserialize(app_id, app_sig_dict, sig_version, database='default')
```

Deserialize a serialized application signature.

**Parameters**

- `app_id (unicode)` – The application ID.
- `app_sig_dict (dict)` – The dictionary containing application signature data.
- `sig_version (int)` – The version of the serialized signature data.
- `database (unicode, optional)` – The name of the database.

**Returns**

The resulting signature instance.

**Return type**

`AppSignature`

**Raises**

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

```
__init__(app_id, legacy_app_label=None, upgrade_method=None, applied_migrations=None)
```

Initialize the signature.

**Parameters**

- `app_id (unicode)` – The ID of the application. This will be the application label. On modern versions of Django, this may differ from the legacy app label.
- `legacy_app_label (unicode, optional)` – The legacy label for the application. This is based on the module name.

- **upgrade\_method** (`unicode`, *optional*) – The upgrade method used for this application. This must be a value from `UpgradeMethod`, or `None`.
- **applied\_migrations** (`set` of `unicode`, *optional*) – The migration names that are applied as of this signature.

**property model\_sigs**

The model signatures stored on the application signature.

**property applied\_migrations**

The set of migration names applied to the app.

**Type**

`set` of `unicode`

**is\_empty()**

Return whether the application signature is empty.

An empty application signature contains no models.

**Returns**

`True` if the signature is empty. `False` if it still has models in it.

**Return type**

`bool`

**add\_model(model)**

Add a model to the application signature.

This will construct a `ModelSignature` and add it to this application signature.

**Parameters**

`model` (`django.db.models.Model`) – The model to create the signature from.

**add\_model\_sig(model\_sig)**

Add a model signature to the application signature.

**Parameters**

`model_sig` (`ModelSignature`) – The model signature to add.

**remove\_model\_sig(model\_name)**

Remove a model signature from the application signature.

**Parameters**

`model_name` (`unicode`) – The name of the model.

**Raises**

`django_evolution.errors.MissingSignatureError` – The model name does not represent a known model signature.

**clear\_model\_sigs()**

Clear all model signatures from the application signature.

**get\_model\_sig(model\_name, required=False)**

Return a model signature for the given model name.

**Parameters**

- `model_name` (`unicode`) – The name of the model.
- `required` (`bool`, *optional*) – Whether the model signature must be present. If `True` and the signature is missing, this will raise an exception.

### Returns

The model signature, if found. If no model signature matches the model name, `None` will be returned.

### Return type

`ModelSignature`

### Raises

`django_evolution.errors.MissingSignatureError` – The model signature was not found, and `required` was `True`.

## `diff(old_app_sig)`

Diff against an older application signature.

This will return a dictionary containing the differences between two application signatures.

### Parameters

`old_app_sig` (`AppSignature`) – The old app signature to diff against.

### Returns

A dictionary in the following form:

```
{  
    'changed': {  
        '<model_name>': <ModelSignature diff>,  
        ...  
    },  
    'deleted': [ <list of deleted model names> ],  
    'meta_changed': {  
        '<prop_name>': {  
            'old': <old value>,  
            'new': <new value>,  
        },  
        ...  
    }  
}
```

Any key lacking a value will be omitted from the diff.

### Return type

`collections.OrderedDict`

### Raises

`TypeError` – The old signature provided was not an `AppSignature`.

## `clone()`

Clone the signature.

### Returns

The cloned signature.

### Return type

`AppSignature`

## `serialize(sig_version=2)`

Serialize application data to a signature dictionary.

### Parameters

`sig_version` (`int, optional`) – The signature version to serialize as. This always defaults to the latest.

**Returns**  
The serialized data.

**Return type**  
`dict`

**Raises**  
`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

**\_\_eq\_\_(other)**  
Return whether two application signatures are equal.

**Parameters**  
`other` (`AppSignature`) – The other application signature.

**Returns**  
True if the application signatures are equal. False if they are not.

**Return type**  
`bool`

**\_\_repr\_\_()**  
Return a string representation of the signature.

**Returns**  
A string representation of the signature.

**Return type**  
`unicode`

**\_\_hash\_\_ = None**

```
class django_evolution.signature.ModelSignature(model_name, table_name, db_tablespace=None,
                                                index_together=[], pk_column=None,
                                                unique_together=[], unique_together_applied=False,
                                                db_table_comment=None)
```

Bases: `BaseSignature`

Signature information for a model.

Model signatures store information on the model and include signatures for its fields and `_meta` attributes.

**classmethod from\_model(model)**  
Create a model signature from a model.  
This will store data on the model and its `_meta` attributes, and create a `FieldSignature` for each field.

**Parameters**  
`model` (`django.db.models.Model`) – The model to create a signature from.

**Returns**  
The signature based on the model.

**Return type**  
`ModelSignature`

**classmethod deserialize(model\_name, model\_sig\_dict, sig\_version, database='default')**  
Deserialize a serialized model signature.

**Parameters**

- `model_name` (`unicode`) – The model name.

- **model\_sig\_dict** (`dict`) – The dictionary containing model signature data.
- **sig\_version** (`int`) – The version of the serialized signature data.
- **database** (`unicode`, *optional*) – The name of the database.

### Returns

The resulting signature instance.

### Return type

`ModelSignature`

### Raises

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

**\_\_init\_\_(model\_name, table\_name, db\_tablespace=None, index\_together=[], pk\_column=None, unique\_together=[], unique\_together\_applied=False, db\_table\_comment=None)**

Initialize the signature.

### Parameters

- **model\_name** (`unicode`) – The name of the model.
- **table\_name** (`unicode`) – The name of the table in the database.
- **db\_tablespace** (`unicode`, *optional*) – The tablespace for the model. This is database-specific.
- **index\_together** (`list of tuple`, *optional*) – A list of fields that are indexed together.
- **pk\_column** (`unicode`, *optional*) – The column for the primary key.
- **unique\_together** (`list of tuple`, *optional*) – The list of fields that are unique together.
- **unique\_together\_applied** (`bool`, *optional*) – Whether the `unique_together` value was applied to the database, rather than simply stored in the signature.

New in version 2.1.3.

- **db\_table\_comment** (`str`, *optional*) – The table comment applied to the database.

New in version 2.3.

### property index\_together

A list of fields that are indexed together.

#### Type

`list`

### property unique\_together

A list of fields that are unique together.

#### Type

`list`

### property field\_sigs

The field signatures on the model signature.

### add\_field(field)

Add a field to the model signature.

This will construct a `FieldSignature` and add it to this model signature.

**Parameters**

**field** (`django.db.models.Field`) – The field to create the signature from.

**add\_field\_sig(*field\_sig*)**

Add a field signature to the model signature.

**Parameters**

**field\_sig** (`FieldSignature`) – The field signature to add.

**remove\_field\_sig(*field\_name*)**

Remove a field signature from the model signature.

**Parameters**

**field\_name** (`unicode`) – The name of the field.

**Raises**

`django_evolution.errors.MissingSignatureError` – The field name does not represent a known field signature.

**get\_field\_sig(*field\_name*, *required=False*)**

Return a field signature for the given field name.

**Parameters**

- **field\_name** (`unicode`) – The name of the field.
- **required** (`bool, optional`) – Whether the model signature must be present. If True and the signature is missing, this will raise an exception.

**Returns**

The field signature, if found. If no field signature matches the field name, `None` will be returned.

**Return type**

`FieldSignature`

**Raises**

`django_evolution.errors.MissingSignatureError` – The model signature was not found, and required was True.

**add\_constraint(*constraint*)**

Add an explicit constraint to the models.

This is only used on Django 2.2 or higher. It corresponds to the `model._meta.constraints` <`django.db.models.Options.constraints`> attribute.

**Parameters**

**constraint** (`django.db.models.BaseConstraint`) – The constraint to add.

**add\_constraint\_sig(*constraint\_sig*)**

Add an explicit constraint signature to the models.

This is only used on Django 2.2 or higher. It corresponds to the `model._meta.constraints` <`django.db.models.Options.constraints`> attribute.

**Parameters**

**constraint\_sig** (`ConstraintSignature`) – The constraint signature to add.

**add\_index(*index*)**

Add an explicit index to the models.

This is only used on Django 1.11 or higher. It corresponds to the `model._meta.indexes` <`django.db.models.Options.indexes`> attribute.

### Parameters

`index` (`django.db.models.Index`) – The index to add.

### `add_index_sig(index_sig)`

Add an explicit index signature to the models.

This is only used on Django 1.11 or higher. It corresponds to the `model._meta.indexes` <`django.db.models.Options.indexes` attribute.

### Parameters

`index_sig` (`IndexSignature`) – The index signature to add.

### `apply_unique_together(unique_together)`

Record an applied unique\_together change to the model.

This will store the new unique together value and set a flag indicating it's been applied to the database.

The flag exists to deal with a situation from old versions of Django Evolution where the unique\_together state was stored in the signature but not applied to the database.

New in version 2.1.3.

### Parameters

`unique_together` (`list`) – The new unique\_together value.

### `has_unique_together_changed(old_model_sig)`

Return whether unique\_together has changed between signatures.

`unique_together` is considered to have changed under the following conditions:

- They are different in value.
- Either the old or new is non-empty (even if equal) and evolving from an older signature from Django Evolution pre-0.7, where unique\_together wasn't applied to the database.

### Parameters

`old_model_sig` (`ModelSignature`) – The old model signature to compare against.

### Returns

True if the value has changed. False if they're considered equal for the purposes of evolution.

### Return type

`bool`

### `diff(old_model_sig)`

Diff against an older model signature.

This will return a dictionary containing the differences in fields and meta information between two signatures.

### Parameters

`old_model_sig` (`ModelSignature`) – The old model signature to diff against.

### Returns

A dictionary in the following form:

```
{  
    'added': [  
        <field name>,  
        ...  
    ]  
}
```

(continues on next page)

(continued from previous page)

```

    ],
    'deleted': [
        <field name>,
        ...
    ],
    'changed': {
        <field name>: <FieldSignature diff>,
        ...
    },
    'meta_changed': [
        <'constraints'>,
        <'indexes'>,
        <'index_together'>,
        <'unique_together'>,
    ],
}

```

Any key lacking a value will be omitted from the diff.

#### Return type

`collections.OrderedDict`

#### Raises

`TypeError` – The old signature provided was not a `ModelSignature`.

### `clone()`

Clone the signature.

#### Returns

The cloned signature.

#### Return type

`ModelSignature`

### `serialize(sig_version=2)`

Serialize model data to a signature dictionary.

#### Parameters

`sig_version (int, optional)` – The signature version to serialize as. This always defaults to the latest.

#### Returns

The serialized data.

#### Return type

`dict`

#### Raises

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

### `__eq__(other)`

Return whether two model signatures are equal.

#### Parameters

`other (ModelSignature)` – The other model signature.

#### Returns

True if the model signatures are equal. False if they are not.

**Return type**  
`bool`

**\_\_repr\_\_()**  
Return a string representation of the signature.

**Returns**  
A string representation of the signature.

**Return type**  
`unicode`

**\_\_hash\_\_ = None**

**class** `django_evolution.signature.ConstraintSignature(name, constraint_type, attrs=None)`  
Bases: `BaseSignature`

Signature information for a explicit constraint.

These indexes were introduced in Django 1.11. They correspond to entries in the `model._meta.indexes` <`dango.db.models.Options.indexes` attribute.

Constraint signatures store information on a constraint on model, including the constraint name, type, and any attribute values needed for constructing the constraint.

**classmethod from\_constraint(constraint)**  
Create a constraint signature from a field.

**Parameters**  
`constraint` (`django.db.models.BaseConstraint`) – The constraint to create a signature from.

**Returns**  
The signature based on the constraint.

**Return type**  
`ConstraintSignature`

**classmethod deserialize(constraint\_sig\_dict, sig\_version, database='default')**  
Deserialize a serialized constraint signature.

**Parameters**

- `constraint_sig_dict` (`dict`) – The dictionary containing constraint signature data.
- `sig_version` (`int`) – The version of the serialized signature data.
- `database` (`unicode`, optional) – The name of the database.

**Returns**  
The resulting signature instance.

**Return type**  
`ConstraintSignature`

**Raises**  
`dango_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

**\_\_init\_\_(name, constraint\_type, attrs=None)**  
Initialize the signature.

**Parameters**

- **name** (`unicode`) – The name of the constraint.
- **constraint\_type** (`cls`) – The class for the constraint. This would be a subclass of `django.db.models.BaseConstraint`.
- **attrs** (`dict`, *optional*) – Attributes to pass when constructing the constraint.

**clone()**

Clone the signature.

**Returns**

The cloned signature.

**Return type**

`ConstraintSignature`

**serialize(sig\_version=2)**

Serialize constraint data to a signature dictionary.

**Parameters**

**sig\_version** (`int`, *optional*) – The signature version to serialize as. This always defaults to the latest.

**Returns**

The serialized data.

**Return type**

`dict`

**Raises**

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

**\_\_eq\_\_(other)**

Return whether two constraint signatures are equal.

**Parameters**

**other** (`ConstraintSignature`) – The other constraint signature.

**Returns**

True if the constraint signatures are equal. False if they are not.

**Return type**

`bool`

**\_\_hash\_\_()**

Return a hash of the signature.

This is required for comparison within a `set`.

**Returns**

The hash of the signature.

**Return type**

`int`

**\_\_repr\_\_()**

Return a string representation of the signature.

**Returns**

A string representation of the signature.

### Return type

`unicode`

`class django_evolution.signature.IndexSignature(fields, name=None, expressions=None, attrs=None)`

Bases: `BaseSignature`

Signature information for an explicit index.

These indexes were introduced in Django 1.11. They correspond to entries in the `model._meta.indexes` <`django.db.models.Options.indexes` attribute.

Changed in version 2.2: Added a new `attrs` attribute for storing:

- `db_tablespace` from Django 2.0+
- `condition` from Django 2.2+
- `include` and `opclasses` from Django 3.2+

Added a new `expressions` attribute for Django 3.2+.

`classmethod from_index(index)`

Create an index signature from an index.

#### Parameters

`index (django.db.models.Index)` – The index to create the signature from.

#### Returns

The signature based on the index.

### Return type

`IndexSignature`

`classmethod deserialize(index_sig_dict, sig_version, database='default')`

Deserialize a serialized index signature.

#### Parameters

- `index_sig_dict (dict)` – The dictionary containing index signature data.
- `sig_version (int)` – The version of the serialized signature data.
- `database (unicode, optional)` – The name of the database.

#### Returns

The resulting signature instance.

### Return type

`IndexSignature`

#### Raises

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

`__init__(fields, name=None, expressions=None, attrs=None)`

Initialize the signature.

#### Parameters

- `fields (list of unicode)` – The list of field names the index is comprised of.
- `name (unicode, optional)` – The optional name of the index.
- `expressions (list, optional)` – A list of expressions for the index.
- `attrs (dict, optional)` – Additional attributes to pass when constructing the index.

**clone()**

Clone the signature.

**Returns**

The cloned signature.

**Return type**

*IndexSignature*

**serialize(sig\_version=2)**

Serialize index data to a signature dictionary.

**Parameters**

**sig\_version** (*int, optional*) – The signature version to serialize as. This always defaults to the latest.

**Returns**

The serialized data.

**Return type**

*dict*

**Raises**

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

**\_\_eq\_\_(other)**

Return whether two index signatures are equal.

**Parameters**

**other** (*IndexSignature*) – The other index signature.

**Returns**

True if the index signatures are equal. False if they are not.

**Return type**

*bool*

**\_\_hash\_\_()**

Return a hash of the signature.

This is required for comparison within a `set`.

**Returns**

The hash of the signature.

**Return type**

*int*

**\_\_repr\_\_()**

Return a string representation of the signature.

**Returns**

A string representation of the signature.

**Return type**

*unicode*

**class django\_evolution.signature.FieldSignature(field\_name, field\_type, fieldAttrs=None, related\_model=None)**

Bases: *BaseSignature*

Signature information for a field.

Field signatures store information on a field on model, including the field name, type, and any attribute values needed for migrating the schema.

### `classmethod from_field(field)`

Create a field signature from a field.

#### Parameters

`field (django.db.models.Field)` – The field to create a signature from.

#### Returns

The signature based on the field.

#### Return type

`FieldSignature`

### `classmethod deserialize(field_name, field_sig_dict, sig_version, database='default')`

Deserialize a serialized field signature.

#### Parameters

- `field_name (unicode)` – The name of the field.
- `field_sig_dict (dict)` – The dictionary containing field signature data.
- `sig_version (int)` – The version of the serialized signature data.
- `database (unicode, optional)` – The name of the database.

#### Returns

The resulting signature instance.

#### Return type

`FieldSignature`

#### Raises

`django_evolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

### `__init__(field_name, field_type, field_attrs=None, related_model=None)`

Initialize the signature.

#### Parameters

- `field_name (unicode)` – The name of the field.
- `field_type (cls)` – The class for the field. This would be a subclass of `django.db.models.Field`.
- `field_attrs (dict, optional)` – Attributes to set on the field.
- `related_model (unicode, optional)` – The full path to a related model.

### `get_attr_value(attr_name, use_default=True)`

Return the value for an attribute.

By default, this will return the default value for the attribute if it's not explicitly set.

#### Parameters

- `attr_name (unicode)` – The name of the attribute.
- `use_default (bool, optional)` – Whether to return the default value for the attribute if it's not explicitly set.

**Returns**

The value for the attribute.

**Return type**

`object`

**`get_attr_default(attr_name)`**

Return the default value for an attribute.

**Parameters**

`attr_name` (`unicode`) – The attribute name.

**Returns**

The default value for the attribute, or `None`.

**Return type**

`object`

**`is_attr_value_default(attr_name)`**

Return whether an attribute is set to its default value.

**Parameters**

`attr_name` (`unicode`) – The attribute name.

**Returns**

`True` if the attribute's value is set to its default value. `False` if it has a custom value.

**Return type**

`bool`

**`__hash__ = None`****`diff(old_field_sig)`**

Diff against an older field signature.

This will return a list of field names that have changed between this field signature and an older one.

**Parameters**

`old_field_sig` (`FieldSignature`) – The old field signature to diff against.

**Returns**

The list of field names.

**Return type**

`list`

**Raises**

`TypeError` – The old signature provided was not a `FieldSignature`.

**`clone()`**

Clone the signature.

**Returns**

The cloned signature.

**Return type**

`FieldSignature`

**`serialize(sig_version=2)`**

Serialize field data to a signature dictionary.

**Parameters**

**sig\_version** (`int`, *optional*) – The signature version to serialize as. This always defaults to the latest.

**Returns**

The serialized data.

**Return type**

`dict`

**Raises**

`djangoevolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

**`__eq__(other)`**

Return whether two field signatures are equal.

**Parameters**

**other** (`FieldSignature`) – The other field signature.

**Returns**

True if the field signatures are equal. False if they are not.

**Return type**

`bool`

**`__repr__()`**

Return a string representation of the signature.

**Returns**

A string representation of the signature.

**Return type**

`unicode`

### `djangoevolution.signature.validate_sig_version(sig_version)`

Validate that a signature version is supported.

**Parameters**

**sig\_version** (`int`) – The version of the signature to validate.

**Raises**

`djangoevolution.errors.InvalidSignatureVersion` – The signature version provided isn't supported.

## Private API

<code>django_evolution.diff</code>	Support for diffing project signatures.
<code>django_evolution.mock_models</code>	Utilities for building mock database models and fields.
<code>django_evolution.mutators</code>	Mutators responsible for applying mutations.
<code>django_evolution.mutators.app_mutator</code>	Mutator that applies changes to an app.
<code>django_evolution.mutators.model_mutator</code>	Mutator that applies changes to a model.
<code>django_evolution.mutators.sql_mutator</code>	Mutator that applies arbitrary SQL to the database.
<code>django_evolution.placeholders</code>	Placeholder objects for hinted evolutions.
<code>django_evolution.support</code>	Constants indicating available Django features.
<code>django_evolution.compat.apps</code>	Compatibility functions for the application registration.
<code>django_evolution.compat.commands</code>	Compatibility module for management commands.
<code>django_evolution.compat.datastructures</code>	Compatibility imports for data structures.
<code>django_evolution.compat.db</code>	Compatibility functions for database-related operations.
<code>django_evolution.compat.models</code>	Compatibility functions for model-related operations.
<code>django_evolution.compat.picklers</code>	Picklers for working with serialized data.
<code>django_evolution.compat.py23</code>	Compatibility functions for Python 2 and 3.
<code>django_evolution.db.common</code>	Common evolution operations backend for databases.
<code>django_evolution.db.mysql</code>	Evolution operations backend for MySQL/MariaDB.
<code>django_evolution.db.postgresql</code>	Evolution operations backend for Postgres.
<code>django_evolution.db.sql_result</code>	Classes for storing SQL statements and Alter Table operations.
<code>django_evolution.db.sqlite3</code>	Evolution operations backend for SQLite.
<code>django_evolution.db.state</code>	Database state tracking for in-progress evolutions.
<code>django_evolution.utils.apps</code>	Utilities for working with apps.
<code>django_evolution.utils.datastructures</code>	Utilities for working with data structures.
<code>django_evolution.utils.evolutions</code>	Utilities for working with evolutions and mutations.
<code>django_evolution.utils.graph</code>	Dependency graphs for tracking and ordering evolutions and migrations.
<code>django_evolution.utils.migrations</code>	Utility functions for working with Django Migrations.
<code>django_evolution.utils.models</code>	Utilities for working with models.
<code>django_evolution.utils.sql</code>	Utilities for working with SQL statements.

### `django_evolution.diff`

Support for diffing project signatures.

Changed in version 2.2: Moved `django_evolution.placeholders.NullFieldInitialCallback` into its own module.

### Classes

<code>Diff(original_project_sig, target_project_sig)</code>	Generates diffs between project signatures.
---	---

`class django_evolution.diff.Diff(original_project_sig, target_project_sig)`

Bases: `object`

Generates diffs between project signatures.

The resulting diff is contained in two attributes:

```

self.changed = {
    app_label: {
        'changed': {
            model_name : {
                'added': [ list of added field names ]
                'deleted': [ list of deleted field names ]
                'changed': {
                    field: [ list of modified property names ]
                },
                'meta_changed': {
                    'constraints': new value
                    'db_table_comment': new value
                    'indexes': new value
                    'index_together': new value
                    'unique_together': new value
                }
            }
        },
        'deleted': [ list of deleted model names ]
    }
}
self.deleted = {
    app_label: [ list of models in deleted app ]
}

```

**`__init__(original_project_sig, target_project_sig)`**

Initialize the object.

**Parameters**

- **original\_project\_sig** (`django_evolution.signature.ProjectSignature`) – The original project signature for the diff.
- **target\_project\_sig** (`django_evolution.signature.ProjectSignature`) – The target project signature for the diff.

**`is_empty(ignore_apps=True)`**

Return whether the diff is empty.

This is used to determine if both signatures are effectively equal. If `ignore_apps` is set, this will ignore changes caused by deleted applications.

**Parameters**

`ignore_apps (bool, optional)` – Whether to ignore changes to the applications list.

**Returns**

True if the diff is empty and signatures are equal. False if there are changes between the signatures.

**Return type**

`bool`

**`__str__()`**

Return a string description of the diff.

This will describe the changes found in the diff, for human consumption.

**Returns**

The string representation of the diff.

**Return type**`unicode`**`evolution()`**

Return the mutations needed for resolving the diff.

This will attempt to return a hinted evolution, consisting of a series of mutations for each affected application. These mutations will convert the database from the original to the target signatures.

**Returns**

An ordered dictionary of mutations. Each key is an application label, and each value is a list of mutations for the application.

**Return type**`collections.OrderedDict`

## **`django_evolution.mock_models`**

Utilities for building mock database models and fields.

### Functions

---

<code>create_field</code> (project_sig, field_name, ...[, ...])	Create a Django field instance for the given signature data.
---	--

---

### Classes

---

<code>MockMeta</code> (project_sig, app_name, model_name, ...)	A mock of a models Options object, based on the model signature.
<code>MockModel</code> (project_sig, app_name, model_name, ...)	A mock model.
<code>MockRelated</code> (related_model, model, field)	A mock RelatedObject for relation fields.

---

`django_evolution.mock_models.create_field`(*project\_sig*, *field\_name*, *field\_type*, *fieldAttrs*, *parentModel*, *relatedModel=None*)

Create a Django field instance for the given signature data.

This creates a field in a way that's compatible with a variety of versions of Django. It takes in data such as the field's name and attributes and creates an instance that can be used like any field found on a model.

**Parameters**

- **`field_name` (`unicode`)** – The name of the field.
- **`field_type` (`cls`)** – The class for the type of field being constructed. This must be a subclass of `django.db.models.Field`.
- **`fieldAttrs` (`dict`)** – Attributes to set on the field.
- **`parentModel` (`cls`)** – The parent model that would own this field. This must be a subclass of `django.db.models.Model`.
- **`relatedModel` (`unicode`, optional)** – The full class path to a model this relates to. This requires a `django.db.models.ForeignKey` field type.

### Returns

A new field instance matching the provided data.

### Return type

`django.db.models.Field`

```
class django_evolution.mock_models.MockMeta(project_sig, app_name, model_name, model_sig,  
                                         managed=False, auto_created=False)
```

Bases: `object`

A mock of a models Options object, based on the model signature.

This emulates the standard Meta class for a model, storing data and providing mock functions for setting up fields from a signature.

```
__init__(project_sig, app_name, model_name, model_sig, managed=False, auto_created=False)
```

Initialize the meta instance.

### Parameters

- **project\_sig** (`django_evolution.signature.ProjectSignature`) – The project's schema signature.
- **app\_name** (`unicode`) – The name of the Django application owning the model.
- **model\_name** (`unicode`) – The name of the model.
- **model\_sig** (`dict`) – The model's schema signature.
- **managed** (`bool, optional`) – Whether this represents a model managed internally by Django, rather than a developer-created model.
- **auto\_created** (`bool, optional`) – Whether this represents an auto-created model (such as an intermediary many-to-many model).

**property local\_fields**

A list of all local fields on the model.

**property fields**

A list of all local fields on the model.

**property local\_many\_to\_many**

A list of all local Many-to-Many fields on the model.

**property label**

A label shown for this model.

New in version 2.2.

**setup\_fields(model, stub=False)**

Set up the fields listed in the model's signature.

For each field in the model signature's list of fields, a field instance will be created and stored in `_fields` or `_many_to_many` (depending on the type of field).

Some fields (for instance, a field representing a primary key) may also influence the attributes on this model.

### Parameters

- **model** (`cls`) – The model class owning this meta instance. This must be a subclass of `django.db.models.Model`.

- **stub** (`bool`, *optional*) – If provided, only a primary key will be set up. This is used internally when creating relationships between models and fields in order to prevent recursive relationships.

**`__getattr__(name)`**

Return an attribute from the meta class.

This will look up the attribute from the correct location, depending on the attribute being accessed.

**Parameters**

`name` (`unicode`) – The attribute name.

**Returns**

The attribute value.

**Return type**

`object`

**`get_field(name)`**

Return a field with the given name.

**Parameters**

`name` (`unicode`) – The name of the field.

**Returns**

The field with the given name.

**Return type**

`django.db.models.Field`

**Raises**

`django.db.models.fields.FieldDoesNotExist` – The field could not be found.

**`get_field_by_name(name)`**

Return information on a field with the given name.

This is a stub that provides only basic functionality. It will return information for a field with the given name, with most data hard-coded.

**Parameters**

`name` (`unicode`) – The name of the field.

**Returns**

A tuple of information for the following:

- The field instance (`django.db.models.Field`)
- The model (hard-coded as `None`)
- Whether this field is owned by this model (hard-coded as `True`)
- Whether this is for a many-to-many relationship (hard-coded as `None`)

**Return type**

`tuple`

**Raises**

`django.db.models.fields.FieldDoesNotExist` – The field could not be found.

```
class django_evolution.mock_models.MockModel(project_sig, app_name, model_name, model_sig,
                                             auto_created=False, managed=False, stub=False,
                                             db_name=None)
```

Bases: `object`

A mock model.

This replicates some of the state and functionality of a model for use when generating, reading, or mutating signatures.

`__init__(project_sig, app_name, model_name, model_sig, auto_created=False, managed=False, stub=False, db_name=None)`

Initialize the model.

### Parameters

- `project_sig` (`djongo_evolution.signature.ProjectSignature`) – The project's schema signature.
- `app_name` (`unicode`) – The name of the Django app that owns the model.
- `model_name` (`unicode`) – The name of the model.
- `model_sig` (`dict`) – The model's schema signature.
- `auto_created` (`bool, optional`) – Whether this represents an auto-created model (such as an intermediary many-to-many model).
- `managed` (`bool, optional`) – Whether this represents a model managed internally by Django, rather than a developer-created model.
- `stub` (`bool, optional`) – Whether this is a stub model. This is used internally to create models that only contain a primary key field and no others, for use when dealing with circular relationships.
- `db_name` (`unicode, optional`) – The name of the database where the model would be read from or written to.

`__repr__()`

Return a string representation of the model.

### Returns

A string representation of the model.

### Return type

`unicode`

`__hash__()`

Return a hash of the model instance.

This is used to allow the model instance to be used as a key in a dictionary.

Django would return a hash of the primary key's value, but that's not necessary for our needs, and we don't have field values in most mock models.

### Returns

The hash of the model.

### Return type

`int`

`__eq__(other)`

Return whether two mock models are equal.

Both are considered equal if they're both mock models with the same app name and model name.

**Parameters**

**other** ([MockModel](#)) – The other mock model to compare to.

**Returns**

True if both are equal. False if they are not.

**Return type**

`bool`

---

```
class django_evolution.mock_models.MockRelated(related_model, model, field)
```

Bases: `object`

A mock RelatedObject for relation fields.

This replicates some of the state and functionality of `django.db.models.related.RelatedObject`, used for generating signatures and applying mutations.

`__init__(related_model, model, field)`

Initialize the object.

**Parameters**

- **related\_model** ([MockModel](#)) – The mock model on the other end of the relation.
- **model** ([MockModel](#)) – The mock model on this end of the relation.
- **field** ([django.db.models.Field](#)) – The field owning the relation.

## [django\\_evolution.mutators](#)

Mutators responsible for applying mutations.

Changed in version 2.2: The classes have all been moved to nested modules. This module will provide forwarding imports, and will continue to be the primary place to import these mutations.

---

<code>AppMutator</code>	Tracks and runs mutations for an app.
<code>ModelMutator</code>	Tracks and runs mutations for a model.
<code>SQLMutator</code>	A mutator that applies arbitrary SQL to the database.

---

### [django\\_evolution.mutators.app\\_mutator](#)

Mutator that applies changes to an app.

New in version 2.2.

## Classes

---

<code>AppMutator(app_label, project_sig, ...[, ...])</code>	Tracks and runs mutations for an app.
---	---------------------------------------

---

```
class django_evolution.mutators.app_mutator.AppMutator(app_label, project_sig, database_state,
                                                       legacy_app_label=None, database=None)
```

Bases: `BaseMutator`

Tracks and runs mutations for an app.

An AppMutator is bound to a particular app name, and handles operations that apply to anything on that app.

This will create a ModelMutator internally for each set of adjacent operations that apply to the same model, allowing the database operations backend to optimize those operations. This means that it's in the best interest of a developer to keep related mutations batched together as much as possible.

After all operations are added, the caller is expected to call `to_sql()` to get the SQL statements needed to apply those operations. Once called, the mutator is finalized, and new operations cannot be added.

Changed in version 2.2: Moved into the `django_evolution.mutators.app_mutator` module.

**classmethod `from_evolver(evolver, app_label, legacy_app_label=None, update_evolver=True)`**

Create an AppMutator based on the state from an Evolver.

#### Parameters

- **evolver** (`django_evolution.evolve.Evolver`) – The Evolver containing the state for the app mutator.
- **app\_label** (`unicode`) – The label of the app to evolve.
- **legacy\_app\_label** (`unicode, optional`) – The legacy label of the app to evolve. This is based on the module name and is used in the transitioning of pre-Django 1.7 signatures.

#### Returns

The new app mutator.

#### Return type

`AppMutator`

**`__init__(app_label, project_sig, database_state, legacy_app_label=None, database=None)`**

Initialize the mutator.

#### Parameters

- **app\_label** (`unicode`) – The label of the app to evolve.
- **project\_sig** (`django_evolution.signature.ProjectSignature`) – The project signature being evolved.
- **database\_state** (`django_evolution.db.state.DatabaseState`) – The database state information to manipulate.
- **legacy\_app\_label** (`unicode, optional`) – The legacy label of the app to evolve. This is based on the module name and is used in the transitioning of pre-Django 1.7 signatures.
- **database** (`unicode, optional`) – The name of the database being evolved.

**`run_mutation(mutation)`**

Runs a mutation that applies to this app.

If the mutation applies to a model, a ModelMutator for that model will be given the job of running this mutation. If the prior operation operated on the same model, then the previously created ModelMutator will be used. Otherwise, a new one will be created.

**`run_mutations(mutations)`**

Runs a list of mutations.

**`add_sql(mutation, sql)`**

Adds SQL that applies to the application.

**`to_sql()`**

Return SQL for the operations added to this mutator.

The SQL will represent all the operations made by the mutator. Once called, no new operations can be added.

## Returns

The list of SQL statements.

Each item may be one of the following:

1. A Unicode string representing an SQL statement
2. A tuple in the form of `(sql_statement, sql_params)`
3. An instance of `django_evolution.db.sql_result.SQLResult`.

## Return type

`list`

## `django_evolution.mutators.model_mutator`

Mutator that applies changes to a model.

New in version 2.2.

## Classes

---

`ModelMutator(app_mutator, model_name)`

Tracks and runs mutations for a model.

---

`class django_evolution.mutators.model_mutator.ModelMutator(app_mutator, model_name)`

Bases: `BaseAppStateMutator`

Tracks and runs mutations for a model.

A `ModelMutator` is bound to a particular model (by type, not instance) and handles operations that apply to that model.

Operations are first registered by mutations, and then later provided to the database's operations backend, where they will be applied to the database.

After all operations are added, the caller is expected to call `to_sql()` to get the SQL statements needed to apply those operations. Once called, the mutator is finalized, and new operations cannot be added.

`ModelMutator` only works with mutations that are instances of `BaseModelFieldMutation`.

This is instantiated by `AppMutator`, and should not be created manually.

Changed in version 2.2: Moved into the `django_evolution.mutators.model_mutator` module.

`__init__(app_mutator, model_name)`

Initialize the mutator.

### Parameters

- `app_mutator` (`AppMutator`) – The app mutator that owns this model mutator.
- `model_name` (`unicode`) – The name of the model being evolved.
- `app_label` (`unicode`) – The label of the app to evolve.
- `legacy_app_label` (`unicode`) – The legacy label of the app to evolve. This is based on the module name and is used in the transitioning of pre-Django 1.7 signatures.
- `project_sig` (`django_evolution.signature.ProjectSignature`) – The project signature being evolved.

- **database\_state** (`django_evolution.db.state.DatabaseState`) – The database state information to manipulate.
- **database** (`unicode`, *optional*) – The name of the database being evolved.

### `property model_sig`

The model signature that this mutator is working with.

#### Type

`django_evolution.signature.ModelSignature`

#### Raises

`django_evolution.errors.EvolutionBaselineMissingError` – The model signature or parent app signature could not be found.

### `create_model()`

Create a mock model instance with the stored information.

This is typically used when calling a mutation's `mutate()` function and passing a model instance, but can also be called whenever a new instance of the model is needed for any lookups.

#### Returns

The resulting mock model.

#### Return type

`django_evolution.mock_models.MockModel`

#### Raises

`django_evolution.errors.EvolutionBaselineMissingError` – The model signature or parent app signature could not be found.

### `add_column(mutation, field, initial)`

Adds a pending Add Column operation.

This will cause `to_sql()` to include SQL for adding the column with the given information to the model.

### `change_column_type(mutation, old_field, new_field, newAttrs)`

Add a pending Change Column Type operation.

This will cause `to_sql()` to include SQL for changing a field to a new type.

#### Parameters

- **mutation** (`django_evolution.mutations.ChangeField`) – The mutation that triggered this column type change.
- **old\_field** (`django.db.models.Field`) – The old field on the model.
- **new\_field** (`django.db.models.Field`) – The new field on the model.
- **newAttrs** (`dict`) – New attributes set in the `ChangeField`.

### `change_column(mutation, field, newAttrs)`

Adds a pending Change Column operation.

This will cause `to_sql()` to include SQL for changing one or more attributes for the given column.

### `delete_column(mutation, field)`

Adds a pending Delete Column operation.

This will cause `to_sql()` to include SQL for deleting the given column.

**delete\_model(*mutation*)**

Adds a pending Delete Model operation.

This will cause `to_sql()` to include SQL for deleting the model.

**change\_meta(*mutation*, *prop\_name*, *new\_value*)**

Adds a pending Change Meta operation.

This will cause `to_sql()` to include SQL for changing a supported attribute in the model's Meta class.

**add\_sql(*mutation*, *sql*)**

Adds an operation for executing custom SQL.

This will cause `to_sql()` to include the provided SQL statements. The SQL should be a list of a statements.

**run\_mutation(*mutation*)**

Run the specified mutation.

The mutation will be provided with a temporary mock instance of a model that can be used for field or meta lookups.

The mutator must be finalized before this can be called.

Once the mutation has been run, it will call `run_simulation()`, applying changes to the database project signature.

**Parameters**

**mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation to run.

**Raises**

`django_evolution.errors.EvolutionBaselineMissingError` – The model signature or parent app signature could not be found.

**to\_sql()**

Returns SQL for the operations added to this mutator.

The SQL will represent all the operations made by the mutator, as determined by the database operations backend.

Once called, no new operations can be added to the mutator.

**finish\_op(*op*)**

Finishes handling an operation.

This is called by the evolution operations backend when it is done with an operation.

Simulations for the operation's associated mutation will be applied, in order to update the signatures for the changes made by the mutation.

**Parameters**

**op** (`dict`) – The operation that has finished.

### `django_evolution.mutators.sql_mutator`

Mutator that applies arbitrary SQL to the database.

New in version 2.2.

#### Classes

---

<code>SQLMutator(mutation, sql)</code>	A mutator that applies arbitrary SQL to the database.
--	---

---

`class django_evolution.mutators.sql_mutator.SQLMutator(mutation, sql)`

Bases: `BaseMutator`

A mutator that applies arbitrary SQL to the database.

This is instantiated by `AppMutator`, and should not be created manually.

Changed in version 2.2: Moved into the `django_evolution.mutators.sql_mutator` module.

`__init__(mutation, sql)`

Initialize the mutator.

#### Parameters

- `mutation` (`django_evolution.mutations.base.BaseMutation`) – The mutation that generated this SQL.
- `sql` (`list`) – The list of SQL statements. See the return type in `to_sql()` for possible values.

`to_sql()`

Return SQL passed to this mutator.

#### Returns

The list of SQL statements.

Each item may be one of the following:

1. A Unicode string representing an SQL statement
2. A tuple in the form of `(sql_statement, sql_params)`
3. An instance of `django_evolution.db.sql_result.SQLResult`.

#### Return type

`list`

### `django_evolution.placeholders`

Placeholder objects for hinted evolutions.

New in version 2.2.

## Classes

<code>BasePlaceholder([app_label, model_name, ...])</code>	A placeholder object for use in generating hints.
<code>NullFieldInitialCallback([app_label, ...])</code>	A placeholder for an initial value for a field.

```
class django_evolution.placeholders.BasePlaceholder(app_label=None, model_name=None,
                                                    field_name=None)
```

Bases: `object`

A placeholder object for use in generating hints.

Placeholder objects provide stand-ins for values that must be hand-added to the evolution file.

New in version 2.2.

**placeholder\_text = None**

The text used in the placeholder.

**Type**

`unicode`

```
__init__(app_label=None, model_name=None, field_name=None)
```

Initialize the object.

**Parameters**

- **app\_label** (`unicode`, *optional*) – The label of the application owning the model.
- **model\_name** (`unicode`, *optional*) – The name of the model owning the field.
- **field\_name** (`unicode`, *optional*) – The name of the field to return an initial value for.

```
__repr__()
```

Return a string representation of the object.

This is used when outputting the value in a hinted evolution.

**Returns**

The placeholder text.

**Return type**

`unicode`

```
__call__()
```

Handle calls on this object.

This will raise an exception stating that the evolution cannot be performed.

**Raises**

`django_evolution.errors.EvolutionException` – An error stating that an explicit initial value must be provided in place of this object.

```
class django_evolution.placeholders.NullFieldInitialCallback(app_label=None,
                                                               model_name=None,
                                                               field_name=None)
```

Bases: `BasePlaceholder`

A placeholder for an initial value for a field.

This is used in place of an initial value in mutations for fields that don't allow NULL values and don't have an explicit initial value set. It will show up in hinted evolutions as <<USER VALUE REQUIRED>> and will fail to evolve.

`placeholder_text = '<<USER VALUE REQUIRED>>'`

The text used in the placeholder.

**Type**

`unicode`

`__call__()`

Handle calls on this object.

This will raise an exception stating that the evolution cannot be performed.

**Raises**

`django_evolution.errors.EvolutionException` – An error stating that an explicit initial value must be provided in place of this object.

## `django_evolution.support`

Constants indicating available Django features.

### Module Attributes

<code>supports_index_together</code>	Index names changed in Django 1.5, with the introduction of <code>index_together</code> .
<code>supports_indexes</code>	Whether new-style Index classes are available.
<code>supports_q_comparison</code>	Whether <code>Q()</code> objects can be directly compared.
<code>supports_f_comparison</code>	Whether <code>F()</code> objects can be directly compared.
<code>supports_constraints</code>	Whether new-style Constraint classes are available.
<code>supports_db_table_comments</code>	Whether database table comments are available.
<code>supports_migrations</code>	Whether built-in support for Django Migrations is present.

### Functions

`supports_index_feature(attr_name)`      Return whether Index supports a specific attribute.

`django_evolution.support.supports_index_together = True`

Index names changed in Django 1.5, with the introduction of `index_together`.

`django_evolution.support.supports_indexes = True`

Whether new-style Index classes are available.

Django 1.11 introduced formal support for defining explicit indexes not bound to a field definition or as part of `index_together/unique_together`.

**Type**

`bool`

`django_evolution.support.supports_q_comparison = True`

Whether `Q()` objects can be directly compared.

Django 2.0 introduced this support.

Type  
bool

`django_evolution.support.supports_f_comparison = True`

Whether F() objects can be directly compared.

Django 2.0 introduced this support.

Type  
bool

`django_evolution.support.supports_constraints = True`

Whether new-style Constraint classes are available.

Django 2.2 introduced formal support for defining explicit constraints not bound to a field definition.

`django_evolution.support.supports_db_table_comments = False`

Whether database table comments are available.

Django 4.2 introduced formal support for setting comments attached to tables.

Support may vary by database backend.

New in version 2.3.

Type  
bool

`django_evolution.support.supports_migrations = True`

Whether built-in support for Django Migrations is present.

This is available in Django 1.7+.

`django_evolution.support.supports_index_feature(attr_name)`

Return whether Index supports a specific attribute.

**Parameters**

`attr_name` (`unicode`) – The name of the attribute.

**Returns**

True if the attribute is supported on this version of Django. False if it is not.

**Return type**

bool

`django_evolution.compat.apps`

Compatibility functions for the application registration.

This provides functions for app registration and lookup. These functions translate to the various versions of Django that are supported.

### Functions

<code>clear_app_cache()</code>	Clear the Django app/models caches.
<code>get_app(app_label[, emptyOK])</code>	Return the app with the given label.
<code>get_apps()</code>	Return the list of all installed apps with models.
<code>is_app_registered(app)</code>	Return whether the app registry is tracking a given app.
<code>register_app(app_label, app)</code>	Register a new app in the registry.
<code>register_app_models(app_label, model_infos)</code>	Register one or more models to a given app.
<code>unregister_app(app_label)</code>	Unregister an app in the registry.
<code>unregister_app_model(app_label, model_name)</code>	Unregister a model with the given name from the given app.

#### `django_evolution.compat.apps.clear_app_cache()`

Clear the Django app/models caches.

This cache is used in Django  $\geq 1.2$  to quickly return results when fetching models. It needs to be cleared when modifying the model registry.

#### `django_evolution.compat.apps.get_app(app_label, emptyOK=False)`

Return the app with the given label.

This returns the app from the app registry on Django  $\geq 1.7$ , and from the old-style cache on Django  $< 1.7$ .

##### **app\_label (str):**

The label for the app containing the models.

##### **emptyOK (bool, optional):**

Impacts the return value if the app has no models in it.

#### **Returns**

The app module, if available.

If the app module is available, but the models module is not and `emptyOK` is set, this will return `None`. Otherwise, if modules are not available, this will raise `ImproperlyConfigured`.

##### **Return type**

`module`

#### **Raises**

`django.core.exceptions.ImproperlyConfigured` – The app module was not found, or it was found but a models module was not and `emptyOK` was `False`.

#### `django_evolution.compat.apps.get_apps()`

Return the list of all installed apps with models.

This returns the apps from the app registry on Django  $\geq 1.7$ , and from the old-style cache on Django  $< 1.7$ .

#### **Returns**

A list of all the modules containing model classes.

##### **Return type**

`list`

## `django_evolution.compat.commands`

Compatibility module for management commands.

### Classes

---

<code>BaseCommand([stdout, stderr, no_color, ...])</code>	Base command compatible with a range of Django versions.
<code>OptionParserWrapper(parser)</code>	Compatibility wrapper for OptionParser.

---

`class django_evolution.compat.commands.OptionParserWrapper(parser)`

Bases: `object`

Compatibility wrapper for OptionParser.

This exports a more modern `ArgumentParser`-based API for `OptionParser`, for use when adding arguments in management commands. This only contains a subset of the functionality of `ArgumentParser`.

`__init__(parser)`

Initialize the wrapper.

#### Parameters

`parser (optparse.OptionParser)` – The option parser.

`add_argument(*args, **kwargs)`

Add an argument to the parser.

This is a simple wrapper that provides compatibility with most of `argparse.ArgumentParser.add_argument()`. It supports the types that `optparse.OptionParser.add_option()` supports (though those types should be passed as the primitive types and not as the string names).

#### Parameters

- `*args (tuple)` – Positional arguments to pass to `optparse.OptionParser.add_option()`.
- `**kwargs (dict)` – Keyword arguments to pass to `optparse.OptionParser.add_option()`.

`class django_evolution.compat.commands.BaseCommand(stdout=None, stderr=None, no_color=False, force_color=False)`

Bases: `BaseCommand`

Base command compatible with a range of Django versions.

This is a version of `django.core.management.base.BaseCommand` that supports the modern way of adding arguments while retaining compatibility with older versions of Django. See the parent class's documentation for details on usage.

`property use_argparse`

Whether argparse should be used for argument parsing.

This is used internally by Django.

`create_parser(*args, **kwargs)`

Create a parser for the command.

This is a wrapper around Django's method that ensures compatibility with old-style ( $\leq 1.6$ ) and new-style ( $\geq 1.7$ ) argument parsing logic.

### Parameters

- **\*args** (`tuple`) – Positional arguments to pass to the parent method.
- **\*\*kwargs** (`dict`) – Keyword arguments to pass to the parent method.

### Returns

The argument parser. This will be a `optparse.OptionParser` or a `argparse.ArgumentParser`.

### Return type

`object`

## `add_arguments(parser)`

Add arguments to the command.

By default, this does nothing. Subclasses can override to add additional arguments.

### Parameters

**parser** (`object`) – The argument parser. This will be a `optparse.OptionParser` or a `argparse.ArgumentParser`.

## `__getattribute__(name)`

Return an attribute from the command.

If the attribute name is “option\_list”, some special work will be done to ensure we’re returning a valid list that the caller can work with, even if the options were created in `add_arguments()`.

### Parameters

**name** (`unicode`) – The attribute name.

### Returns

The attribute value.

### Return type

`object`

## `django_evolution.compat.datastructures`

Compatibility imports for data structures.

This provides imports for data structures that are needed internally, to provide compatibility with different versions of Django.

### `class django_evolution.compat.datastructures.OrderedDict`

Bases: `dict`

Dictionary that remembers insertion order

#### `__init__(*args, **kwargs)`

#### `__setitem__(key, value, /)`

Set `self[key]` to `value`.

#### `__delitem__(key, /)`

Delete `self[key]`.

#### `__iter__()`

Implement `iter(self)`.

**`__reversed__()`** <==> `reversed(od)`

**`clear()`** → None. Remove all items from od.

**`popitem(last=True)`**  
 Remove and return a (key, value) pair from the dictionary.  
 Pairs are returned in LIFO order if last is true or FIFO order if false.

**`move_to_end(key, last=True)`**  
 Move an existing element to the end (or beginning if last is false).  
 Raise KeyError if the element does not exist.

**`__sizeof__()`** → size of D in memory, in bytes

**`update([E], **F)`** → None. Update D from dict/iterable E and F.  
 If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

**`keys()`** → a set-like object providing a view on D's keys

**`items()`** → a set-like object providing a view on D's items

**`values()`** → an object providing a view on D's values

**`__ne__(value, /)`**  
 Return self!=value.

**`pop(k[, d])`** → v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised.

**`setdefault(key, default=None)`**  
 Insert key with a value of default if key is not in the dictionary.  
 Return the value for key if key is in the dictionary, else default.

**`__repr__()`**  
 Return repr(self).

**`__reduce__()`**  
 Return state information for pickling

**`copy()`** → a shallow copy of od

**`fromkeys(value=None)`**  
 Create a new ordered dictionary with keys from iterable and values set to value.

**`__eq__(value, /)`**  
 Return self==value.

**`__ge__(value, /)`**  
 Return self>=value.

**`__gt__(value, /)`**  
 Return self>value.

**`__hash__ = None`**

```
__le__(value, /)
    Return self<=value.

__lt__(value, /)
    Return self<value.
```

### django\_evolution.compat.db

Compatibility functions for database-related operations.

This provides functions for database operations, SQL generation, index name generation, and more. These functions translate to the various versions of Django that are supported.

#### Functions

<code>atomic(using)</code>	Perform database operations atomically within a transaction.
<code>collect_sql_schema_editor(connection)</code>	Create a schema editor for the purpose of collecting SQL.
<code>convert_table_name(connection, name)</code>	Convert a table name to a format required by the database backend.
<code>create_constraint_name(connection, r_col, ...)</code>	Return the name of a constraint.
<code>create_index_name(connection, table_name[, ...])</code>	Return the name for an index for a field.
<code>create_index_together_name(connection, ...)</code>	Return the name of an index for an index_together.
<code>db_get_installable_models_for_app(app, db_state)</code>	Return models that can be installed in a database.
<code>db_routerAllows_migrate(database, ...)</code>	Return whether a database router allows migrate operations for a model.
<code>db_routerAllows_schema_upgrade(database, ...)</code>	Return whether a database router allows a schema upgrade for a model.
<code>db_routerAllows_syncdb(database, model_cls)</code>	Return whether a database router allows syncdb operations for a model.
<code>digest(connection, *args)</code>	Return a digest hash for a set of arguments.
<code>sql_add_constraints(connection, model, refs)</code>	Return SQL statements for adding constraints.
<code>sql_create_app(app[, db_name])</code>	Return SQL statements for creating all models for an app.
<code>sql_create_for_many_to_many_field(...)</code>	Return SQL statements for creating a ManyToManyField's table.
<code>sql_create_models(models[, tables, db_name, ...])</code>	Return SQL statements for creating a list of models.
<code>sql_delete(app[, db_name])</code>	Return SQL statements for deleting all models in an app.
<code>sql_delete_constraints(connection, model, ...)</code>	Return SQL statements for deleting constraints.
<code>sql_delete_index(connection, model, index_name)</code>	Return SQL statements for deleting an index.
<code>sql_indexes_for_field(connection, model, field)</code>	Return SQL statements for creating indexes for a field.
<code>sql_indexes_for_fields(connection, model, fields)</code>	Return SQL statements for creating indexes covering multiple fields.
<code>sql_indexes_for_model(connection, model)</code>	Return SQL statements for creating all indexes for a model.

`django_evolution.compat.db.atomic(using=None)`

Perform database operations atomically within a transaction.

The caller can use this to ensure SQL statements are executed within a transaction and then cleaned up nicely if there's an error.

This provides compatibility with all supported versions of Django.

**Parameters**

- **using** (`str`, *optional*) – The database connection name to use. Defaults to the default database connection.

`django_evolution.compat.db.create_constraint_name(connection, r_col, col, r_table, table)`

Return the name of a constraint.

This provides compatibility with all supported versions of Django.

**Parameters**

- **connection** (`object`) – The database connection.
- **r\_col** (`str`) – The column name for the source of the relation.
- **col** (`str`) – The column name for the “to” end of the relation.
- **r\_table** (`str`) – The table name for the source of the relation.
- **table** (`str`) – The table name for the “to” end of the relation.

**Returns**

The generated constraint name for this version of Django.

**Return type**

`str`

`django_evolution.compat.db.create_index_name(connection, table_name, field_names=[], col_names=[], unique=False, suffix='')`

Return the name for an index for a field.

This provides compatibility with all supported versions of Django.

**Parameters**

- **connection** (`object`) – The database connection.
- **table\_name** (`str`) – The name of the table.
- **field\_names** (`list of str`, *optional*) – The list of field names for the index.
- **col\_names** (`list of str`, *optional*) – The list of column names for the index.
- **unique** (`bool`, *optional*) – Whether or not this index is unique.
- **suffix** (`str`, *optional*) – A suffix for the index. This is only used with Django >= 1.7.

**Returns**

The generated index name for this version of Django.

**Return type**

`str`

`django_evolution.compat.db.create_index_together_name(connection, table_name, field_names)`

Return the name of an index for an index\_together.

This provides compatibility with all supported versions of Django >= 1.5. Prior versions don't support index\_together.

**Parameters**

- **connection** (`object`) – The database connection.

- **table\_name** (`str`) – The name of the table.
- **field\_names** (`list` of `str`) – The list of field names indexed together.

### Returns

The generated index name for this version of Django.

### Return type

`str`

`djangoevolution.compat.db.db_get_installable_models_for_app(app, db_state)`

Return models that can be installed in a database.

### Parameters

- **app** (`module`) – The models module for the app.
- **db\_state** (`djangoevolution.db.state.DatabaseState`) – The introspected state of the database.

`djangoevolution.compat.db.db_routerAllowsMigrate(database, app_label, model_cls)`

Return whether a database router allows migrate operations for a model.

This will only return True for Django 1.7 and newer and if the router allows migrate operations. This is compatible with both the Django 1.7 and 1.8+ versions of `allow_migrate`.

### Parameters

- **database** (`unicode`) – The name of the database.
- **app\_label** (`unicode`) – The application label.
- **model\_cls** (`type`) – The model class.

### Returns

True if routers allow migrate for this model.

### Return type

`bool`

`djangoevolution.compat.db.db_routerAllowsSchemaUpgrade(database, app_label, model_cls)`

Return whether a database router allows a schema upgrade for a model.

This is a convenience wrapper around `db_routerAllowsMigrate()` and `db_routerAllowsSyncdb()`.

### Parameters

- **database** (`unicode`) – The name of the database.
- **app\_label** (`unicode`) – The application label.
- **model\_cls** (`type`) – The model class.

### Returns

True if routers allow migrate for this model.

### Return type

`bool`

`djangoevolution.compat.db.db_routerAllowsSyncdb(database, model_cls)`

Return whether a database router allows syncdb operations for a model.

This will only return True for Django 1.6 and older and if the router allows syncdb operations.

### Parameters

- **database** (`unicode`) – The name of the database.
- **model\_cls** (`type`) – The model class.

**Returns**

True if routers allow syncdb for this model.

**Return type**

`bool`

`djangoevolution.compat.db.digest(connection, *args)`

Return a digest hash for a set of arguments.

This is mostly used as part of the index/constraint name generation processes. It offers compatibility with a range of Django versions.

**Parameters**

- **connection** (`object`) – The database connection.
- **\*args** (`tuple`) – The positional arguments used to build the digest hash out of.

**Returns**

The resulting digest hash.

**Return type**

`str`

`djangoevolution.compat.db.sql_add_constraints(connection, model, refs)`

Return SQL statements for adding constraints.

This provides compatibility with all supported versions of Django.

**Parameters**

- **connection** (`object`) – The database connection.
- **model** (`django.db.models.Model`) – The database model to add constraints on.
- **refs** (`dict`) – A dictionary of constraint references to add.

The keys are instances of `django.db.models.Model`. The values are a tuple of (`django.db.models.Model`, `django.db.models.Field`).

**Warning:** Keys may be removed as constraints are added. Make sure to pass in a copy of the dictionary if the original dictionary must be preserved.

**Returns**

The list of SQL statements for adding constraints.

**Return type**

`list`

`djangoevolution.compat.db.sql_create_app(app, db_name=None)`

Return SQL statements for creating all models for an app.

This provides compatibility with all supported versions of Django.

**Parameters**

- **app** (`module`) – The application module.

- **db\_name** (`str`, *optional*) – The database connection name. Defaults to the default database connection.

### Returns

The list of SQL statements used to create the models for the app.

### Return type

`list`

```
django_evolution.compat.db.sql_create_models(models, tables=None, db_name=None,  
                                              return_deferred=False)
```

Return SQL statements for creating a list of models.

This provides compatibility with all supported versions of Django.

It's recommended that callers include auto-created models in the list, to ensure all references are correct.

Changed in version 2.2: Added the ```return_deferred``` argument.

### Parameters

- **models** (`list` of `type`) – The list of `Model` subclasses.
- **tables** (`list` of `unicode`, *optional*) – A list of existing table names from the database. If not provided, this will be introspected from the database.
- **db\_name** (`str`, *optional*) – The database connection name. Defaults to the default database connection.
- **return\_deferred** (`bool`, *optional*) – Whether to return any deferred SQL separately from the model creation SQL. If True, the return type will change to a tuple.

### Returns

If `return_deferred=False` (the default), this will be a list of SQL statements used to create the models for the app.

If `return_deferred=True`, this will be a 2-tuple in the form of `(list_of_sql, list_of_deferred_sql)`.

### Return type

`list` or `tuple`

```
django_evolution.compat.db.sql_create_for_many_to_many_field(connection, model, field)
```

Return SQL statements for creating a ManyToManyField's table.

This provides compatibility with all supported versions of Django.

### Parameters

- **connection** (`object`) – The database connection.
- **model** (`djongo.db.models.Model`) – The model for the ManyToManyField's relations.
- **field** (`djongo.db.models.ManyToManyField`) – The field setting up the many-to-many relation.

### Returns

The list of SQL statements for creating the table and constraints.

### Return type

`list`

`django_evolution.compat.db.sql_delete(app, db_name=None)`

Return SQL statements for deleting all models in an app.

This provides compatibility with all supported versions of Django.

#### Parameters

- **app** (`module`) – The application module containing the models to delete.
- **db\_name** (`str, optional`) – The database connection name. Defaults to the default database connection.

#### Returns

The list of SQL statements for deleting the models and constraints.

#### Return type

`list`

`django_evolution.compat.db.sql_delete_constraints(connection, model, remove_refs)`

Return SQL statements for deleting constraints.

This provides compatibility with all supported versions of Django.

#### Parameters

- **connection** (`object`) – The database connection.
- **model** (`djongo.db.models.Model`) – The database model to delete constraints on.
- **remove\_refs** (`dict`) – A dictionary of constraint references to remove.

The keys are instances of `djongo.db.models.Model`. The values are a tuple of (`djongo.db.models.Model, djongo.db.models.Field`).

**Warning:** Keys may be removed as constraints are deleted. Make sure to pass in a copy of the dictionary if the original dictionary must be preserved.

#### Returns

The list of SQL statements for deleting constraints.

#### Return type

`list`

`django_evolution.compat.db.sql_delete_index(connection, model, index_name)`

Return SQL statements for deleting an index.

This provides compatibility with all supported versions of Django.

#### Parameters

- **connection** (`object`) – The database connection.
- **model** (`djongo.db.models.Model`) – The database model to delete an index on.
- **index\_name** (`unicode`) – The name of the index to delete.

#### Returns

The list of SQL statements for deleting the index.

#### Return type

`list`

`django_evolution.compat.db.sql_indexes_for_field(connection, model, field)`

Return SQL statements for creating indexes for a field.

This provides compatibility with all supported versions of Django.

### Parameters

- **connection** (`object`) – The database connection.
- **model** (`django.db.models.Model`) – The database model owning the field.
- **field** (`django.db.models.Field`) – The field being indexed.

### Returns

The list of SQL statements for creating the indexes.

### Return type

`list`

`django_evolution.compat.db.sql_indexes_for_fields(connection, model, fields, index_together=False)`

Return SQL statements for creating indexes covering multiple fields.

This provides compatibility with all supported versions of Django.

### Parameters

- **connection** (`object`) – The database connection.
- **model** (`django.db.models.Model`) – The database model owning the fields.
- **fields** (`list` of `django.db.models.Field`) – The list of fields for the index.
- **index\_together** (`bool`, optional) – Whether this is from an index\_together rule.

### Returns

The list of SQL statements for creating the indexes.

### Return type

`list`

`django_evolution.compat.db.sql_indexes_for_model(connection, model)`

Return SQL statements for creating all indexes for a model.

This provides compatibility with all supported versions of Django.

### Parameters

- **connection** (`object`) – The database connection.
- **model** (`django.db.models.Model`) – The database model to create indexes for.

### Returns

The list of SQL statements for creating the indexes.

### Return type

`list`

`django_evolution.compat.db.truncate_name(identifier, length=None, hash_len=4)`

Shorten a SQL identifier to a repeatable mangled version with the given length.

If a quote stripped name contains a namespace, e.g. `USERNAME`.`TABLE`, truncate the table portion only.

## django\_evolution.compat.models

Compatibility functions for model-related operations.

This provides functions for working with models or importing moved fields. These translate to the various versions of Django that are supported.

### Functions

<code>get_field_is_hidden(field)</code>	Return whether a field is hidden.
<code>get_field_is_many_to_many(field)</code>	Return whether a field is a Many-to-Many field.
<code>get_field_is_relation(field)</code>	Return whether a field is a relation.
<code>get_model_name(model)</code>	Return the model's name.
<code>get_models([app_mod, include_auto_created])</code>	Return the models belonging to an app.
<code>get_rel_target_field(field)</code>	Return the target field for a field's relation.
<code>get_remote_field(field)</code>	Return the remote field for a relation.
<code>get_remote_field_model(rel)</code>	Return the model a relation is pointing to.
<code>get_remote_field_related_model(rel)</code>	Return the model a relation is pointing from.
<code>set_model_name(model, name)</code>	Set the name of a model.

### exception django\_evolution.compat.models.FieldDoesNotExist

Bases: `Exception`

The requested model field does not exist

```
class django_evolution.compat.models.GenericForeignKey(ct_field='content_type', fk_field='object_id',
                                                       for_concrete_model=True)
```

Bases: `FieldCacheMixin`

Provide a generic many-to-one relation through the `content_type` and `object_id` fields.

This class also doubles as an accessor to the related object (similar to `ForwardManyToOneDescriptor`) by adding itself as a model attribute.

`auto_created = False`

`concrete = False`

`hidden = False`

`is_relation = True`

`many_to_many = False`

`many_to_one = True`

`one_to_many = False`

`one_to_one = False`

`related_model = None`

`remote_field = None`

```
__init__(ct_field='content_type', fk_field='object_id', for_concrete_model=True)
```

```
editable = False

contribute_to_class(cls, name, **kwargs)

get_filter_kwargs_for_object(obj)
    See corresponding method on Field

get_forward_related_filter(obj)
    See corresponding method on RelatedField

__str__()
    Return str(self).

check(**kwargs)

get_cache_name()

get_content_type(obj=None, id=None, using=None)

get_prefetch_queryset(instances, queryset=None)

__get__(instance, cls=None)

__set__(instance, value)

class django_evolution.compat.models.GenericRelation(to, object_id_field='object_id',
                                                       content_type_field='content_type',
                                                       for_concrete_model=True,
                                                       related_query_name=None,
                                                       limit_choices_to=None, **kwargs)

Bases: ForeignObject

Provide a reverse to a relation created by a GenericForeignKey.

auto_created = False

many_to_many = False

many_to_one = False

one_to_many = True

one_to_one = False

rel_class
    alias of GenericRel

mti_inherited = False

__init__(to, object_id_field='object_id', content_type_field='content_type', for_concrete_model=True,
        related_query_name=None, limit_choices_to=None, **kwargs)

check(**kwargs)

resolve_related_fields()

get_path_info(filtered_relation=None)
    Get path from this field to the related model.
```

**get\_reverse\_path\_info(*filtered\_relation=None*)**  
Get path from the related model to this field's model.

**value\_to\_string(*obj*)**  
Return a string value of this field from the passed obj. This is used by the serialization framework.

**contribute\_to\_class(*cls, name, \*\*kwargs*)**  
Register the field with the model class it belongs to.  
If private\_only is True, create a separate instance of this field for every subclass of cls, even if cls is not an abstract model.

**set\_attributes\_from\_rel()**

**get\_internal\_type()**

**get\_content\_type()**  
Return the content type associated with this field's model.

**get\_extra\_restriction(*where\_class, alias, remote\_alias*)**  
Return a pair condition used for joining and subquery pushdown. The condition is something that responds to as\_sql(compiler, connection) method.  
Note that currently referring both the ‘alias’ and ‘related\_alias’ will not work in some conditions, like subquery pushdown.  
A parallel method is get\_extra\_descriptor\_filter() which is used in instance.fieldname related object fetching.

**bulk\_related\_objects(*objs, using='default'*)**  
Return all objects related to objs via this GenericRelation.

**django\_evolution.compat.models.get\_field\_is\_hidden(*field*)**  
Return whether a field is hidden.  
New in version 2.2.

**Parameters**  
**field** (django.db.models.Field) – The field to check.

**Returns**  
True if the field is hidden. False if it is not.

**Return type**  
bool

**django\_evolution.compat.models.get\_field\_is\_many\_to\_many(*field*)**  
Return whether a field is a Many-to-Many field.  
New in version 2.2.

**Parameters**  
**field** (django.db.models.Field) – The field to check.

**Returns**  
True if the field is a Many-to-Many field. False if it is not.

**Return type**  
bool

### `django_evolution.compat.models.get_field_is_relation(field)`

Return whether a field is a relation.

A field is a relation if it's an object like a `django.db.models.ForeignKey` or `django.db.models.ManyToManyField`, or if it's a relation utility field like `django.db.models.fields.related.ForeignObjectRel` or `django.db.models.fields.related.ManyToOneRel`.

New in version 2.2.

#### Parameters

`field (django.db.models.Field or :class:`django.db.models.RelatedField`)` – The field to check.

#### Returns

True if the field is a relation. False if it is not.

#### Return type

`bool`

### `django_evolution.compat.models.get_model(app_label, model_name=None, require_ready=True)`

Return the model matching the given app\_label and model\_name.

As a shortcut, app\_label may be in the form <app\_label>.<model\_name>.

model\_name is case-insensitive.

Raise `LookupError` if no application exists with this label, or no model exists with this name in the application.  
Raise `ValueError` if called with a single argument that doesn't contain exactly one dot.

### `django_evolution.compat.models.get_models(app_mod=None, include_auto_created=False)`

Return the models belonging to an app.

#### Parameters

- `app_mod (module, optional)` – The application module.
- `include_auto_created (bool, optional)` – Whether to return auto-created models (such as many-to-many models) in the results.

#### Returns

The list of modules belonging to the app.

#### Return type

`list`

### `django_evolution.compat.models.get_model_name(model)`

Return the model's name.

#### Parameters

`model (django.db.models.Model)` – The model for which to return the name.

#### Returns

The model's name.

#### Return type

`str`

### `django_evolution.compat.models.get_rel_target_field(field)`

Return the target field for a field's relation.

**Warning:** Despite the name, this should only be called on a `ForeignKey` and not on a relation, in order to avoid consistency issues in the data returned on Django  $\geq 1.7$ .

**Parameters**

**field** (`django.db.models.Field`) – The relation field.

**Returns**

The field on the other end of the relation.

**Return type**

`django.db.models.Field`

`django_evolution.compat.models.get_remote_field(field)`

Return the remote field for a relation.

This will be an intermediary field, such as:

- `django.db.models.fields.related.ForeignObjectRel`
- `django.db.models.fields.related.ManyToOneRel`
- `django.db.models.fields.related.OneToOneRel`
- `django.db.models.fields.related.ManyToManyFieldRel`

This is equivalent to `rel` prior to Django 1.9 and `remote_field` in 1.9 onward.

Changed in version 2.2: On Django  $< 1.9$ , a main relation field (like `django.db.models.ForeignKey`) will return the utility relation, matching the behavior on  $\geq 1.9$ .

**Parameters**

**field** (`django.db.models.Field`) – The relation field.

**Returns**

The remote field on the relation.

**Return type**

`django.db.models.Field`

`django_evolution.compat.models.get_remote_field_model(rel)`

Return the model a relation is pointing to.

This is equivalent to `rel.to` prior to Django 1.9 and `remote_field.model` in 1.9 onward.

**Parameters**

**rel** (`object`) – The relation object. This is expected to be the result of a `get_remote_field()` call.

**Returns**

The model the relation points to. This should be a subclass of `django.db.models.Model()`.

**Return type**

`type`

`django_evolution.compat.models.get_remote_field_related_model(rel)`

Return the model a relation is pointing from.

New in version 2.2.

**Parameters**

**rel** (`object`) – The relation object. This is expected to be the result of a `get_remote_field()` call.

### Returns

The model the relation points to. This should be a subclass of `django.db.models.Model()`.

### Return type

`type`

`django_evolution.compat.models.set_model_name(model, name)`

Set the name of a model.

### Parameters

- `model` (`django.db.models.Model`) – The model to set the new name on.
- `name` (`str`) – The new model name.

## `django_evolution.compat.picklers`

Picklers for working with serialized data.

### Classes

<code>DjangoCompatUnpickler(file, *[...])</code>	Unpickler compatible with changes to Django class/module paths.
<code>SortedDict(*args, **kwargs)</code>	Compatibility for unpickling a <code>SortedDict</code> .

`class django_evolution.compat.picklers.SortedDict(*args, **kwargs)`

Bases: `dict`

Compatibility for unpickling a `SortedDict`.

Old signatures may use an old Django `SortedDict` structure, which does not exist in modern versions. This changes any construction of this data structure into a `collections.OrderedDict`.

`static __new__(cls, *args, **kwargs)`

Construct an instance of the class.

### Parameters

- `*args` (`tuple`) – Positional arguments to pass to the constructor.
- `**kwargs` (`dict`) – Keyword arguments to pass to the constructor.

### Returns

The new instance.

### Return type

`collections.OrderedDict`

`class django_evolution.compat.picklers.DjangoCompatUnpickler(file, *, fix_imports=True, encoding='ASCII', errors='strict')`

Bases: `_Unpickler`

Unpickler compatible with changes to Django class/module paths.

This provides compatibility across Django versions for various field types, updating referenced module paths for fields to a standard location so that the fields can be located on all Django versions.

**`find_class(module, name)`**

Return the class for a given module and class name.

If looking up a class from `django.db.models.fields`, the class will instead be looked up from `django.db.models`, fixing lookups on some Django versions.

**Parameters**

- **module** (`unicode`) – The module path.
- **name** (`unicode`) – The class name.

**Returns**

The resulting class.

**Return type**

`type`

**Raises**

`AttributeError` – The class could not be found in the module.

**`django_evolution.compat.py23`**

Compatibility functions for Python 2 and 3.

**Functions**

<code>pickle.dumps(obj)</code>	Return a pickled representation of an object.
<code>pickle.loads(pickled_str)</code>	Return the unpickled data from a pickle payload.

**`django_evolution.compat.py23.pickle.dumps(obj)`**

Return a pickled representation of an object.

This will always use Pickle protocol 0, which is the default on Python 2, for compatibility across Python 2 and 3.

**Parameters**

- **obj** (`object`) – The object to dump.

**Returns**

The Unicode pickled representation of the object, safe for storing in the database.

**Return type**

`unicode`

**`django_evolution.compat.py23.pickle.loads(pickled_str)`**

Return the unpickled data from a pickle payload.

**Parameters**

- **pickled\_str** (`bytes`) – The pickled data.

**Returns**

The unpickled data.

**Return type**

`object`

### django\_evolution.db.common

Common evolution operations backend for databases.

#### Classes

---

<code>BaseEvolutionOperations(database_state[, ...])</code>	Base class for evolution operations for a database back-end.
---	--

---

```
class django_evolution.db.common.BaseEvolutionOperations(database_state, connection=<django.db.DefaultConnectionProxy object>)
```

Bases: `object`

Base class for evolution operations for a database backend.

**name = None**

The name of the database type.

New in version 2.3.

**supported\_change\_attrs = {'db\_column', 'db\_index', 'db\_table', 'decimal\_places', 'max\_digits', 'max\_length', 'null', 'unique'}**

A set of attributes that can be changed in the database.

**supported\_change\_meta = {'constraints': True, 'db\_table\_comment': False, 'index\_together': True, 'indexes': True, 'unique\_together': True}**

**mergeable\_ops = ('add\_column', 'change\_column', 'change\_metadef\_delete\_column')**

**ignored\_m2mAttrs = {<class 'django.db.models.fields.related.ManyToManyField'>: {'null'}}**

**default\_tablespace = None**

The default tablespace for the database, if tablespaces are supported.

New in version 2.2.

**Type**

`unicode`

**change\_column\_type\_setsAttrs = True**

Whether a column type change operation also sets new attributes.

If `False`, attributes will be set through the standard field change operation.

New in version 2.2.

**Type**

`bool`

**alter\_table\_sql\_result\_cls**

alias of `AlterTableSQLResult`

**`__init__(database_state, connection=<django.db.DefaultConnectionProxy object>)`**

Initialize the evolution operations.

**Parameters**

- **database\_state** (`djongo_evolution.db.state.DatabaseState`) – The database state to track information through.
- **connection** (`object`) – The database connection.

**`can_add_index(index)`**

Return whether an index can be added to this database.

This will determine if the database connection supports the state represented in the index well enough to be written to the database.

Note that not all features of an index are required. At the moment, to comply with Django's logic (`BaseDatabaseSchemaEditor.add_index()`), an index can be written so long as it either does not contain expressions or the database backend supports expression indexes.

**Parameters**

`index(django.db.models.Index)` – The index that would be written.

**Returns**

True if the index can be written. False if it cannot.

**Return type**

`bool`

**`get_field_typeAllowsDefault(field)`**

Return whether default values are allowed for a field.

By default, default values are always allowed. Subclasses should override this if some types do not allow for defaults.

New in version 2.2.

**Parameters**

`field(django.db.models.Field)` – The field to check.

**Returns**

True if default values are allowed. False if they're not.

**Return type**

`bool`

**`get_deferrable_sql()`**

Return the SQL for marking a reference as deferrable.

New in version 2.2.

**Returns**

The SQL for marking a reference as deferrable.

**Return type**

`unicode`

**`get_change_column_type_sql(model, old_field, new_field)`**

Return SQL for changing a column type.

This should be limited to the `ALTER TABLE` or equivalent for changing the column. It should not affect constraints or other fields.

Subclasses must implement this, unless they override `change_column_type()`.

New in version 2.2.

### Parameters

- **model** (`type`) – The parent model of the column.
- **old\_field** (`django.db.models.Field`) – The old field being replaced.
- **new\_field** (`django.db.models.Field`) – The new replacement field.

### Returns

The SQL statements for changing the column type.

### Return type

`django_evolution.db.sql_result.SQLResult`

**build\_column\_schema**(*model*, *field*, *initial=None*, *skip\_null\_constraint=False*,  
*skip\_primary\_or\_unique\_constraint=False*, *skip\_references=False*)

Return information on the schema for building a column.

This is used when creating or re-creating columns on a table.

### Parameters

- **model** (`type`) – The parent model of the column.
- **field** (`django.db.models.Field`) – The field to build the column schema from.
- **initial** (`object` or `callable`) – The initial data for the column.
- **skip\_null\_constraint** (`bool`, *optional*) – Whether to skip adding NULL/NOT NULL constraints. This can be used to temporarily omit this part of the schema while adding the column.
- **skip\_references** (`bool`, *optional*) – Whether to skip adding REFERENCES ... information. This can be used to temporarily omit this part of the schema while adding the column, handling that step separately.

### Returns

The schema information. This has the following keys:

#### **db\_type** (`unicode`):

The database-specific column type.

#### **definition** (`list`):

A list of parts of the column schema definition. Each of these is a keyword (which may or may not have spaces) or values used for constructing the column.

#### **definition\_sql\_params** (`list`):

The list of SQL parameters to pass to the executor. These will be safely handled by the database backend.

#### **name** (`unicode`):

The name of the column.

### Return type

`dict`

**generate\_table\_ops\_sql**(*mutator*, *ops*)

Generates SQL for a sequence of mutation operations.

This will process each operation one-by-one, generating default SQL, using `generate_table_op_sql()`.

**generate\_table\_op\_sql**(*mutator, op, prev\_sql\_result, prev\_op*)

Generates SQL for a single mutation operation.

This will call different SQL-generating functions provided by the class, depending on the details of the operation.

If two adjacent operations can be merged together (meaning that they can be turned into one ALTER TABLE statement), they'll be placed in the same AlterTableSQLResult.

**quote\_sql\_param**(*param*)

Add protective quoting around an SQL string parameter

**rename\_column**(*model, old\_field, new\_field*)

Renames the specified column.

This must be implemented by subclasses. It must return an SQLResult or AlterTableSQLResult representing the SQL needed to rename the column.

**get\_rename\_table\_sql**(*model, old\_db\_table, new\_db\_table*)

Return SQL for renaming a table.

**Parameters**

- **model** (`djangodb.models.Model`) – The model representing the table to rename.
- **old\_db\_table** (`unicode`) – The old table name.
- **new\_db\_table** (`unicode`) – The new table name.

**Returns**

The resulting SQL for renaming the table.

**Return type**

`djangoevolution.db.sql_result.SQLResult`

**rename\_table**(*model, old\_db\_table, new\_db\_table*)

Rename a table.

This will take care of removing and then restoring any primary field constraints. If an evolver backend doesn't support this, or has another method for managing these constraints, it should override this method.

**Parameters**

- **model** (`djangodb.models.Model`) – The model representing the table to rename.
- **old\_db\_table** (`unicode`) – The old table name.
- **new\_db\_table** (`unicode`) – The new table name.

**Returns**

The resulting SQL for renaming the table.

**Return type**

`djangoevolution.db.sql_result.SQLResult`

**delete\_column**(*model, f*)**delete\_table**(*table\_name*)**add\_m2m\_table**(*model, field*)

Return SQL statements for creating a ManyToManyField's table.

**Parameters**

- **model** (`djangodb.models.Model`) – The database model owning the field.

- **field** (`django.db.models.ManyToManyField`) – The field owning the table.

### Returns

The list of SQL statements for creating the table.

### Return type

`list`

## `add_column(model, field, initial)`

Add a column to a table.

### Parameters

- **model** (`type`) – The model representing the table the column will be added to.
- **field** (`django.db.models.Field`) – The field representing the column being added.
- **initial** (`object` or `callable`) – The initial data to set for the column in all rows. If this is a callable, it will be called and the result will be used.

### Returns

The SQL for adding the column.

### Return type

`django_evolution.db.sql_result.AlterTableSQLResult`

## `set_field_null(model, field, null)`

## `create_index(model, field)`

Returns the SQL for creating an index for a single field.

The index will be recorded in the database signature for future operations within the transaction, and the appropriate SQL for creating the index will be returned.

This is not intended to be overridden.

## `create_unique_index(model, index_name, fields)`

## `drop_index(model, field)`

Returns the SQL for dropping an index for a single field.

The index matching the field's column will be looked up and, if found, the SQL for dropping it will be returned.

If the index was not found on the database or in the database signature, this won't return any SQL statements.

This is not intended to be overridden. Instead, subclasses should override `get_drop_index_sql`.

## `drop_index_by_name(model, index_name)`

Returns the SQL to drop an index, given an index name.

The index will be removed from the database signature, and the appropriate SQL for dropping the index will be returned.

This is not intended to be overridden. Instead, subclasses should override `get_drop_index_sql`.

## `get_drop_index_sql(model, index_name)`

Returns the database-specific SQL to drop an index.

This can be overridden by subclasses if they use a syntax other than "DROP INDEX <name>;"

**get\_new\_index\_name**(model, fields, unique=False)

Return a newly generated index name.

This returns a unique index name for any indexes created by django-evolution, based on how Django would compute the index.

**Parameters**

- **model** (`django.db.models.Model`) – The database model for the index.
- **fields** (`list` of `django.db.models.Field`) – The list of fields for the index.
- **unique** (`bool`, optional) – Whether this index is unique.

**Returns**

The generated name for the index.

**Return type**

`str`

**get\_new\_constraint\_name**(table\_name, column)

Return a newly-generated constraint name.

**Parameters**

- **table\_name** (`unicode`) – The name of the table.
- **column** (`unicode`) – The name of the column.

**Returns**

The new constraint name.

**Return type**

`unicode`

**get\_default\_index\_name**(table\_name, field)

Return a default index name for the database.

This will return an index name for the given field that matches what the database or Django database backend would automatically generate when marking a field as indexed or unique.

This can be overridden by subclasses if the database or Django database backend provides different values.

**Parameters**

- **table\_name** (`str`) – The name of the table for the index.
- **field** (`django.db.models.Field`) – The field for the index.

**Returns**

The name of the index.

**Return type**

`str`

**get\_default\_index\_together\_name**(table\_name, fields)

Returns a default index name for an index\_together.

This will return an index name for the given field that matches what Django uses for index\_together fields.

**Parameters**

- **table\_name** (`str`) – The name of the table for the index.
- **fields** (`list` of `django.db.models.Field`) – The fields for the index.

### Returns

The name of the index.

### Return type

`str`

`change_column_attrs(model, mutation, field_name, newAttrs)`

Return the SQL for changing one or more column attributes.

This will generate all the statements needed for changing a set of attributes for a column.

The resulting `AlterTableSQLResult` contains all the SQL needed to apply these attributes.

### Parameters

- `model` (`type`) – The model class that owns the field.
- `mutation` (`django_evolution.mutations.BaseModelMutation`) – The mutation applying this change.
- `field_name` (`unicode`) – The name of the field on the model.
- `newAttrs` (`dict`) – A dictionary mapping attributes to new values.

### Returns

The SQL for modifying the column.

### Return type

`django_evolution.db.sql_result.AlterTableSQLResult`

`change_column_attr_null(model, mutation, field, old_value, new_value)`

Returns the SQL for changing a column's NULL/NOT NULL attribute.

`change_column_attr_decimal_type(model, mutation, field, new_max_digits, new_decimal_places)`

Return SQL for changing a column's max digits and decimal places.

This is used for `DecimalField` and subclasses to change the maximum number of digits or decimal places. As these are used together as a column type, they must be considered together as one attribute change.

### Parameters

- `model` (`type`) – The model class that owns the field.
- `mutation` (`django_evolution.mutations.BaseModelMutation`) – The mutation applying this change.
- `field` (`django.db.models.DecimalField`) – The field being modified.
- `new_max_digits` (`int`) – The new value for `max_digits`. If `None`, it wasn't provided in the attribute change.
- `new_decimal_places` (`int`) – The new value for `decimal_places`. If `None`, it wasn't provided in the attribute change.

### Returns

The SQL for modifying the value.

### Return type

`django_evolution.db.sql_result.AlterTableSQLResult`

`change_column_attr_max_length(model, mutation, field, old_value, new_value)`

Returns the SQL for changing a column's max length.

**`change_column_attr_db_column`**(model, mutation, field, old\_value, new\_value)

Returns the SQL for changing a column's name.

**`change_column_attr_db_table`**(model, mutation, field, old\_value, new\_value)

Returns the SQL for changing the table for a ManyToManyField.

**`change_column_attrs_db_index_unique`**(model, mutation, field, old\_db\_index, new\_db\_index, old\_unique, new\_unique)

Return SQL for changing indexes due to db\_index or unique.

This determines whether standard or unique indexes need to be dropped or added, and returns the resulting SQL.

Unique indexes are dropped if a field went from unique=True to unique=False.

If not dropping a unique index, but the field was set to db\_index=True, unique=False, and either db\_index=False or unique=True is being set, a standard index will be dropped.

Unique indexes are added if a field went from unique=False to unique=True.

If not adding a unique index, but the field was set to db\_index=False or unique=True and is being set to db\_index=True, unique=False, then a standard index will be added.

New in version 2.3.

**Parameters**

- **model** (`djangoevolution.db.models.Model`) – The model being changed.
- **mutation** (`djangoevolution.mutations.BaseModelMutation`) – The mutation applying this change.
- **field** (`djangoevolution.db.models.DecimalField`) – The field being modified.
- **old\_db\_index** (`bool`) – The old db\_index value.
- **new\_db\_index** (`bool`) – The new db\_index value.
- **old\_unique** (`bool`) – The old unique value.
- **new\_unique** (`bool`) – The new unique value.

**Returns**

The SQL for dropping and/or adding indexes.

**Return type**

`djangoevolution.db.sql_result.AlterTableSQLResult`

**`change_column_attr_db_index`**(model, mutation, field, old\_value, new\_value)

Return the SQL for creating/dropping indexes for a column.

If setting db\_index=True, SQL for generating the index will be returned.

If setting db\_index=False, SQL for dropping the index will be returned.

Creating or dropping the SQL will also modify the cached/queued database index state, used by other operations that work with indexes.

Subclasses should override this if they're sensitive to the order in which SQL is generated or cached/queued database index state is modified.

**Parameters**

- **model** (`djangoevolution.db.models.Model`) – The model being changed.

- **mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation applying this change.
- **field** (`django.db.models.DecimalField`) – The field being modified.
- **old\_value** (`bool`) – The old value for `db_index`.
- **new\_value** (`bool`) – The new value for `db_index`.

### Returns

The resulting SQL for creating the index or scheduling a drop.

### Return type

`django_evolution.db.sql_result.SQLResult`

**change\_column\_attr\_unique**(*model*, *mutation*, *field*, *old\_value*, *new\_value*)

Returns the SQL to change a field's unique flag.

Changing the unique flag for a given column will affect indexes. If setting unique to True, an index will be created in the database signature for future operations within the transaction. If False, the index will be dropped from the database signature.

The SQL needed to change the column will be returned.

This is not intended to be overridden. Instead, subclasses should override `get_change_unique_sql`.

**change\_column\_type**(*model*, *old\_field*, *new\_field*, *newAttrs*)

Return SQL to change the type of a column.

New in version 2.2.

### Parameters

- **model** (`type`) – The type of model owning the field.
- **old\_field** (`django.db.models.Field`) – The old field.
- **new\_field** (`django.db.models.Field`) – The new field.
- **newAttrs** (`dict`) – New attributes set in the [ChangeField](#).

### Returns

The SQL statements for changing the column type.

### Return type

`django_evolution.sql_result.AlterTableSQLResult`

**get\_change\_unique\_sql**(*model*, *field*, *new\_unique\_value*, *constraint\_name*, *initial*)

Returns the database-specific SQL to change a column's unique flag.

This can be overridden by subclasses if they use a different syntax.

**get\_drop\_unique\_constraint\_sql**(*model*, *index\_name*)

**change\_meta\_unique\_together**(*model*, *old\_unique\_together*, *new\_unique\_together*)

Change the unique\_together constraints of a table.

### Parameters

- **model** (`django.db.models.Model`) – The model being changed.
- **old\_unique\_together** (`list`) – The old value for `unique_together`.
- **new\_unique\_together** (`list`) – The new value for `unique_together`.

**Returns**

The SQL statements for changing the `unique_together` constraints.

**Return type**

`django_evolution.sql_result.SQLResult`

**change\_meta\_index\_together**(*model*, *old\_index\_together*, *new\_index\_together*)

Change the `index_together` indexes of a table.

**Parameters**

- **model** (`django.db.models.Model`) – The model being changed.
- **old\_index\_together** (`list`) – The old value for `index_together`.
- **new\_index\_together** (`list`) – The new value for `index_together`.

**Returns**

The SQL statements for changing the `index_together` indexes.

**Return type**

`django_evolution.sql_result.SQLResult`

**change\_meta\_db\_table\_comment**(*model*, *old\_comment*, *new\_comment*)

Change the comment for a table.

Table comments are supported for some database backends in Django 4.2 and higher. This generates the SQL for setting a comment for a given table.

New in version 2.3.

**Parameters**

- **model** (`django.db.models.Model`) – The model being changed.
- **old\_comment** (`unicode`) – The old comment.
- **new\_comment** (`unicode`) – The new comment.

**Returns**

The SQL statements for changing `Meta.db_table_comment`.

**Return type**

`django_evolution.sql_result.SQLResult`

**change\_meta\_constraints**(*model*, *old\_constraints*, *new\_constraints*)

Change the constraints of a table.

Constraints are a feature available in Django 2.2+ that allow for defining custom constraints on a table on `Meta.constraints`.

This will calculate the old and new list of constraint instances, and the list of added/removed constraints, and call out to `get_update_table_constraints_sql()` to generate the SQL for changing them.

**Parameters**

- **model** (`django.db.models.Model`) – The model being changed.
- **old\_constraints** (`list of dict`) – A serialized representation of the old value for `Meta.constraints`.  
This will contain `name` and `type` keys, as well as all attributes on the constraint.
- **new\_constraints** (`list of dict`) – A serialized representation of the new value for `Meta.constraints`.

This is in the same format as `old_constraints`.

#### Returns

The SQL statements for changing `Meta.constraints`.

#### Return type

`django_evolution.sql_result.SQLResult`

`get_update_table_constraints_sql(model, old_constraints, new_constraints, to_add, to_remove)`

Return SQL for updating the constraints on a table.

The generated SQL will remove any old constraints and add any new constraints.

By default, this uses the schema editor for the connection. Subclasses can modify this if they need custom logic.

#### Parameters

- `model` (`django.db.models.Model`) – The model being changed.
- `old_constraints` (`list` of `:class:`~ django.db.models.constraints.BaseConstraint``) – The old constraints pre-evolution.
- `new_constraints` (`list` of `:class:`~ django.db.models.constraints.BaseConstraint``) – The new constraints post-evolution.
- `to_add` (`list` of `django.db.models.constraints.BaseConstraint`) – A list of new constraints to add to the database that weren't set before.
- `to_remove` (`list` of `django.db.models.constraints.BaseConstraint`) – A list of old constraints to remove from the database that aren't set now.

#### Returns

The SQL statements for changing the constraints.

#### Return type

`django_evolution.sql_result.SQLResult`

`change_meta_indexes(model, old_indexes, new_indexes)`

Change the indexes of a table defined in a model's indexes list.

This will apply a set of indexes serialized from a `Meta.indexes` to the database. The serialized values are those passed to `ChangeMeta`, in the form of:

```
[  
    {  
        'condition': {<deconstructed>}},  
        'db_tablespace': '...',  
        'expressions': [{<deconstructed>}], ...],  
        'fields': ['field1', '-field2_sorted_desc'],  
        'include': ['...'], ...],  
        'name': 'optional-index-name',  
        'opclasses': ['...', ...],  
    },  
    ...  
]
```

**Parameters**

- **model** (`django.db.models.Model`) – The model being changed.
- **old\_indexes** (`list`) – The old serialized value for the indexes.
- **new\_indexes** (`list`) – The new serialized value for the indexes.

**Returns**

The SQL statements for changing the indexes.

**Return type**

`django_evolution.sql_result.SQLResult`

**`get_fields_for_names(model, field_names, allow_sort_prefixes=False)`**

Return the field instances for the given field names.

This will go through each of the provided field names, optionally handling a sorting prefix (-, used by Django 1.11+'s `Index` field lists), and return the field instance for each.

**Parameters**

- **model** (`django.db.models.Model`) – The model to fetch fields from.
- **field\_names** (`list` of `unicode`) – The list of field names to fetch.
- **allow\_sort\_prefixes** (`bool`, *optional*) – Whether to allow sorting prefixes in the field names.

**Returns**

The resulting list of fields.

**Return type**

`list` of `django.db.models.Field`

**`get_column_names_for_fields(fields)`****`get_constraints_for_table(table_name)`**

Return all known constraints/indexes on a table.

This will scan the table for any constraints or indexes. It generally will wrap Django's database introspection support if available (on Django >= 1.7), falling back on in-house implementations on earlier releases.

New in version 2.2.

**Parameters**

**table\_name** (`unicode`) – The name of the table.

**Returns**

A dictionary mapping index names to a dictionary containing:

**columns** (`list`):

The list of columns that the index covers.

**unique** (`bool`):

Whether this is a unique index.

**Return type**

`dict`

### `get_indexes_for_table(table_name)`

Return all known indexes on a table.

This is a fallback used only on Django 1.6, due to lack of proper introspection on that release. It should only be called internally by `get_constraints_for_table()`.

#### Parameters

`table_name (unicode)` – The name of the table.

#### Returns

A dictionary mapping index names to a dictionary containing:

##### `columns (list):`

The list of columns that the index covers.

##### `unique (bool):`

Whether this is a unique index.

#### Return type

`dict`

### `stash_field_ref_constraints(model, replaced_fields={}, renamed_db_tables={})`

Return SQL for removing constraints on a primary key field.

This should be called before performing an operation that renames a field or changes the table on a ManyToManyField on databases that support adding/dropping constraints on primary keys. The constraints can then be restored through `restore_field_ref_constraints()`.

As of Django Evolution 2.0, this only considers fields on ManyToManyFields defined by `model`, keeping behavior consistent with prior versions of Django Evolution.

#### Parameters

- `model (django.db.models.Model)` – The model owning the fields to remove constraints from.
- `replaced_fields (dict)` – A dictionary mapping old fields to new fields. Each field is expected to be a primary key. These will be checked for field name and column changes.
- `renamed_db_tables (dict)` – A dictionary mapping old table names to new table names. This is used when renaming many-to-many intermediary tables.

#### Returns

A tuple containing the following items:

1. The SQLResult that contains the SQL to remove the current constraints.
2. A dictionary containing internal stashed state for restoring constraints. This should be considered opaque.

#### Return type

`tuple`

### `restore_field_ref_constraints(stash)`

Return SQL for adding back field constraints on a table.

This should be called after performing an operation that renames a field or a ManyToMany table name on databases that support adding/dropping constraints on primary keys.

This requires a prior call to `stash_field_ref_constraints()`.

#### Parameters

`stash (dict)` – Stashed constraint data from `stash_field_ref_constraints()`.

**Returns**

The SQL statements for adding back constraints on the field.

**Return type**

`django_evolution.sql_result.SQLResult`

**normalize\_initial(*initial*)**

Normalize an initial value.

If the value is callable, it will be called and the result will be used. If that result is a string, it will be assumed to be something safe for embedding directly into SQL.

Anything else is considered best used as a SQL parameter.

New in version 2.3.

**Parameters**

**initial** (`object` or `callable`) – The initial value to normalize.

**Returns**

A 2-tuple of:

1. The normalized initial value.
2. Whether it can be embedded directly into SQL. If `False`, it should be used in SQL query parameter list.

**Return type**

`tuple`

**normalize\_value(*value*)****normalize\_bool(*value*)**

## **django\_evolution.db.mysql**

Evolution operations backend for MySQL/MariaDB.

### **Classes**

---

`EvolutionOperations(database_state[, connection])` Evolution operations for MySQL and MariaDB databases.

---

`class django_evolution.db.mysql.EvolutionOperations(database_state, connection=<django.db.DefaultConnectionProxy object>)`

Bases: `BaseEvolutionOperations`

Evolution operations for MySQL and MariaDB databases.

`name = 'MySQL / MariaDB'`

The name of the database type.

New in version 2.3.

### `get_field_typeAllowsDefault(field)`

Return whether default values are allowed for a field.

New in version 2.2.

#### Parameters

`field (django.db.models.Field)` – The field to check.

#### Returns

True if default values are allowed. False if they’re not.

#### Return type

`bool`

### `get_change_column_type_sql(model, old_field, new_field)`

Return SQL to change the type of a column.

New in version 2.2.

#### Parameters

- `model (type)` – The type of model owning the field.
- `old_field (django.db.models.Field)` – The old field.
- `new_field (django.db.models.Field)` – The new field.

#### Returns

The SQL statements for changing the column type.

#### Return type

`dango_evolution.sql_result.AlterTableSQLResult`

### `delete_column(model, f)`

### `rename_column(model, old_field, new_field)`

Rename the specified column.

This will rename the column through `ALTER TABLE ... CHANGE COLUMN`.

Any constraints on the column will be stashed away before the `ALTER TABLE` and restored afterward.

If the column has not actually changed, or it’s not a real column (a many-to-many relation), then this will return empty statements.

#### Parameters

- `model (type)` – The model representing the table containing the column.
- `old_field (django.db.models.Field)` – The old field definition.
- `new_field (django.db.models.Field)` – The new field definition.

#### Returns

The statements for renaming the column. This may be an empty list if the column won’t be renamed.

#### Return type

`dango_evolution.db.sql_result.AlterTableSQLResult or list`

### `set_field_null(model, field, null)`

### `change_column_attr_max_length(model, mutation, field, old_value, new_value)`

Returns the SQL for changing a column’s max length.

**get\_drop\_index\_sql**(model, index\_name)

Returns the database-specific SQL to drop an index.

This can be overridden by subclasses if they use a syntax other than “DROP INDEX <name>;”

**get\_change\_unique\_sql**(model, field, new\_unique\_value, constraint\_name, initial)

Returns the database-specific SQL to change a column’s unique flag.

This can be overridden by subclasses if they use a different syntax.

**get\_rename\_table\_sql**(model, old\_db\_table, new\_db\_table)

Return SQL for renaming a table.

**Parameters**

- **model** (`django.db.models.Model`) – The model representing the table to rename.
- **old\_db\_table** (`unicode`) – The old table name.
- **new\_db\_table** (`unicode`) – The new table name.

**Returns**

The resulting SQL for renaming the table.

**Return type**

`django_evolution.db.sql_result.SQLResult`

**get\_default\_index\_name**(table\_name, field)

Return a default index name for the database.

This will return an index name for the given field that matches what the database or Django database backend would automatically generate when marking a field as indexed or unique.

This can be overridden by subclasses if the database or Django database backend provides different values.

**Parameters**

- **table\_name** (`str`) – The name of the table for the index.
- **field** (`django.db.models.Field`) – The field for the index.

**Returns**

The name of the index.

**Return type**

`str`

**get\_indexes\_for\_table**(table\_name)

Return all known indexes on a table.

This is a fallback used only on Django 1.6, due to lack of proper introspection on that release.

**Parameters**

- **table\_name** (`unicode`) – The name of the table.

**Returns**

A dictionary mapping index names to a dictionary containing:

**columns** (`list`):

The list of columns that the index covers.

**unique** (`bool`):

Whether this is a unique index.

**Return type**  
dict

### `django_evolution.db.postgresql`

Evolution operations backend for Postgres.

## Classes

---

`EvolutionOperations(database_state[, connection])` Evolution operations for Postgres databases.

---

`class django_evolution.db.postgresql.EvolutionOperations(database_state, connection=<django.db.DefaultConnectionProxy object>)`

Bases: `BaseEvolutionOperations`

Evolution operations for Postgres databases.

`name = 'Postgres'`

The name of the database type.

New in version 2.3.

`default_tablespace = 'pg_default'`

The default tablespace for the database, if tablespaces are supported.

New in version 2.2.

**Type**

unicode

`change_column_type_setsAttrs = False`

Whether a column type change operation also sets new attributes.

If `False`, attributes will be set through the standard field change operation.

New in version 2.2.

**Type**

bool

`alter_field_type_map = {'bigserial': 'bigint', 'serial': 'integer', 'smallserial': 'smallint'}`

A mapping of field types for use when altering types.

New in version 2.2.

`get_change_column_type_sql(model, old_field, new_field)`

Return SQL to change the type of a column.

New in version 2.2.

#### Parameters

- `model` (`type`) – The type of model owning the field.
- `old_field` (`django.db.models.Field`) – The old field.
- `new_field` (`django.db.models.Field`) – The new field.

**Returns**

The SQL statements for changing the column type.

**Return type**

`django_evolution.sql_result.AlterTableSQLResult`

**rename\_column**(*model*, *old\_field*, *new\_field*)

Renames the specified column.

This must be implemented by subclasses. It must return an `SQLResult` or `AlterTableSQLResult` representing the SQL needed to rename the column.

**get\_drop\_unique\_constraint\_sql**(*model*, *index\_name*)

**get\_default\_index\_name**(*table\_name*, *field*)

Return a default index name for the database.

This will return an index name for the given field that matches what the database or Django database backend would automatically generate when marking a field as indexed or unique.

This can be overridden by subclasses if the database or Django database backend provides different values.

**Parameters**

- **table\_name** (`str`) – The name of the table for the index.
- **field** (`djongo.db.models.Field`) – The field for the index.

**Returns**

The name of the index.

**Return type**

`str`

**get\_indexes\_for\_table**(*table\_name*)

Return all known indexes on a table.

This is a fallback used only on Django 1.6, due to lack of proper introspection on that release.

**Parameters**

**table\_name** (`unicode`) – The name of the table.

**Returns**

A dictionary mapping index names to a dictionary containing:

**columns** (`list`):

The list of columns that the index covers.

**unique** (`bool`):

Whether this is a unique index.

**Return type**

`dict`

**normalize\_bool**(*value*)

**change\_column\_attr\_decimal\_type**(*model*, *mutation*, *field*, *new\_max\_digits*, *new\_decimal\_places*)

Return SQL for changing a column's max digits and decimal places.

This is used for `DecimalField` and subclasses to change the maximum number of digits or decimal places. As these are used together as a column type, they must be considered together as one attribute change.

**Parameters**

- **model** (`type`) – The model class that owns the field.
- **mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation applying this change.
- **field** (`django.db.models.DecimalField`) – The field being modified.
- **new\_max\_digits** (`int`) – The new value for `max_digits`. If `None`, it wasn't provided in the attribute change.
- **new\_decimal\_places** (`int`) – The new value for `decimal_places`. If `None`, it wasn't provided in the attribute change.

### Returns

The SQL for modifying the value.

### Return type

`django_evolution.db.sql_result.AlterTableSQLResult`

## `django_evolution.db.sql_result`

Classes for storing SQL statements and Alter Table operations.

### Classes

<code>AlterTableSQLResult(evolver, model[, ...])</code>	Represents one or more SQL statements or Alter Table rules.
<code>SQLResult([sql, pre_sql, post_sql])</code>	Represents one or more SQL statements.

`class django_evolution.db.sql_result.SQLResult(sql=None, pre_sql=None, post_sql=None)`

Bases: `object`

Represents one or more SQL statements.

This is returned by functions generating SQL statements. It can store the main SQL statements to execute, or SQL statements to be executed before or after the main statements.

SQLResults can easily be added together or converted into a flat list of SQL statements to execute.

`__init__(sql=None, pre_sql=None, post_sql=None)`

`add(sql_or_result)`

Adds a list of SQL statements or an SQLResult.

If an SQLResult is passed, its `pre_sql`, `sql`, and `post_sql` lists will be added to this one.

If a list of SQL statements is passed, it will be added to this SQLResult's `sql` list.

#### Parameters

`sql_or_result` (`object`) – The SQL to add. This may be one of the following:

- Another instance of `SQLResult`
- A list of SQL statements
- A single SQL statement
- A tuple pair containing the SQL statement and arguments for that statement
- A function to call later when executing SQL statements

**Raises**

**TypeError** – `sql_or_result` wasn't a supported type.

**`add_pre_sql(sql_or_result)`**

Adds a list of SQL statements or an SQLResult to `pre_sql`.

If an SQLResult is passed, it will be converted into a list of SQL statements.

**`add_sql(sql_or_result)`**

Adds a list of SQL statements or an SQLResult to `sql`.

If an SQLResult is passed, it will be converted into a list of SQL statements.

**`add_post_sql(sql_or_result)`**

Adds a list of SQL statements or an SQLResult to `post_sql`.

If an SQLResult is passed, it will be converted into a list of SQL statements.

**`normalize_sql(sql_or_result)`**

Normalizes a list of SQL statements or an SQLResult into a list.

If a list of SQL statements is provided, it will be returned. If an SQLResult is provided, it will be converted into a list of SQL statements and returned.

**`to_sql()`**

Flattens the SQLResult into a list of SQL statements.

**`__repr__()`**

Return `repr(self)`.

**class** `django_evolution.db.sql_result.AlterTableSQLResult(evolver, model, alter_table=None, *args, **kwargs)`

Bases: `SQLResult`

Represents one or more SQL statements or Alter Table rules.

This is returned by functions generating SQL statements. It can store the main SQL statements to execute, or SQL statements to be executed before or after the main statements.

SQLResults can easily be added together or converted into a flat list of SQL statements to execute.

**`__init__(evolver, model, alter_table=None, *args, **kwargs)`****`add(sql_result)`**

Adds a list of SQL statements or an SQLResult.

If an SQLResult is passed, its `pre_sql`, `sql`, and `post_sql` lists will be added to this one.

If an AlterTableSQLResult is passed, its `alter_table` lists will also be added to this one.

If a list of SQL statements is passed, it will be added to this SQLResult's `sql` list.

**`add_alter_table(alter_table)`**

Adds a list of Alter Table rules to `alter_table`.

**`to_sql()`**

Flattens the AlterTableSQLResult into a list of SQL statements.

Any `alter_table` entries will be collapsed together into ALTER TABLE statements.

**`__repr__()`**

Return `repr(self)`.

### `django_evolution.db.sqlite3`

Evolution operations backend for SQLite.

#### Classes

<code>EvolutionOperations(database_state[, connection])</code>	Evolution operations backend for SQLite.
<code>SQLiteAlterTableSQLResult(evolver, model[, ...])</code>	Represents SQL statements used to rebuild a table on SQLite.

```
class django_evolution.db.sqlite3.SQLiteAlterTableSQLResult(evolver, model, alter_table=None,
                                                               *args, **kwargs)
```

Bases: `AlterTableSQLResult`

Represents SQL statements used to rebuild a table on SQLite.

Unlike most databases, SQLite doesn't offer typical ALTER TABLE support, instead requiring a full table rebuild and data transfer. This class handles that process, allowing operations for the rebuild (adding, deleting, or changing columns) to be batched together.

The rebuild uses the step-by-step instructions recommended by SQLite. It creates a new table with the desired schema, copies all data from the old table, drops the old table, and then renames the new table over.

It can also update the newly-populated rows in the new table with new initial data, if needed by a new column.

**to\_sql()**

Return a list of SQL statements for the table rebuild.

Any `alter_table` operations will be collapsed together into a single table rebuild.

#### Returns

The list of SQL statements to run for the rebuild.

#### Return type

`list of unicode`

```
class django_evolution.db.sqlite3.EvolutionOperations(database_state, connec-
                                                       tion=<django.db.DefaultConnectionProxy
                                                       object>)
```

Bases: `BaseEvolutionOperations`

Evolution operations backend for SQLite.

**name = 'SQLite'**

The name of the database type.

New in version 2.3.

```
supported_change_meta = {'constraints': True, 'db_table_comment': False,
                           'index_together': True, 'indexes': True, 'unique_together': True}
```

**alter\_table\_sql\_result\_cls**

alias of `SQLiteAlterTableSQLResult`

**get\_deferrable\_sql()**

Return the SQL for marking a reference as deferrable.

Despite SQLite3 supporting this, the Django SQLite3 backend does not implement the standard function (`BaseDatabaseOperations.deferrable_sql()`) for this.

This provides a value used internally for building references.

New in version 2.2.

**Returns**

The SQL for marking a reference as deferrable.

**Return type**

`unicode`

**rename\_table(model, old\_db\_table, new\_db\_table)**

Rename a table.

**Parameters**

- **model** (`django_evolution.mock_models.MockModel`) – The model representing the table to rename.
- **old\_db\_table** (`unicode`) – The old table name.
- **new\_db\_table** (`unicode`) – The new table name.

**Returns**

The resulting SQL for renaming the table.

**Return type**

`django_evolution.db.sql_result.SQLResult`

**delete\_column(model, field)**

Delete a column from the table.

**Parameters**

- **model** (`type`) – The `Model` class representing the table to delete the column from.
- **field** (`django.db.models.Field`) – The field representing the column to delete.

**Returns**

The resulting SQL for rebuilding the table.

**Return type**

`django_evolution.db.sql_result.SQLResult`

**rename\_column(model, old\_field, new\_field)**

Rename a column on a table.

**Parameters**

- **model** (`type`) – The `Model` class representing the table to rename the column on.
- **old\_field** (`django.db.models.Field`) – The field representing the old column.
- **new\_field** (`django.db.models.Field`) – The field representing the new column.

**Returns**

The resulting SQL for rebuilding the table.

**Return type**

`django_evolution.db.sql_result.SQLResult`

### `add_column(model, field, initial)`

Add a column to the table.

#### Parameters

- **model** (`type`) – The `Model` class representing the table to add the column to.
- **field** (`django.db.models.Field`) – The field representing the column to add.
- **initial** (`object`) – The initial data to set for the column. If `None`, the data will not be set.

This will be required for NOT NULL columns.

#### Returns

The resulting SQL for rebuilding the table.

#### Return type

`django_evolution.db.sql_result.SQLResult`

### `change_column_attr_null(model, mutation, field, old_value, new_value)`

Change a column's NULL flag.

#### Parameters

- **model** (`type`) – The `Model` class representing the table to change the column on.
- **mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation making this change.
- **field** (`django.db.models.Field`) – The field representing the column to change.
- **old\_value** (`bool`, *unused*) – The old null flag.
- **new\_value** (`bool`) – The new null flag.

#### Returns

The resulting SQL for rebuilding the table.

#### Return type

`django_evolution.db.sql_result.SQLResult`

### `change_column_attr_decimal_type(model, mutation, field, new_max_digits, new_decimal_places)`

Return SQL for changing a column's decimal\_places attribute.

This is used for `DecimalField` and subclasses to change the maximum number of digits or decimal places. As these are used together as a column type, they must be considered together as one attribute change.

#### Parameters

- **model** (`type`) – The model class that owns the field.
- **mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation applying this change.
- **field** (`django.db.models.DecimalField`) – The field being modified.
- **new\_max\_digits** (`int`) – The new value for `max_digits`. If `None`, it wasn't provided in the attribute change.
- **new\_decimal\_places** (`int`) – The new value for `decimal_places`. If `None`, it wasn't provided in the attribute change.

#### Returns

The SQL for modifying the value.

**Return type**

`django_evolution.db.sql_result.AlterTableSQLResult`

**change\_column\_attr\_max\_length**(*model*, *mutation*, *field*, *old\_value*, *new\_value*)

Change a column's max length.

**Parameters**

- **model** (`type`) – The `Model` class representing the table to change the column on.
- **mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation making this change.
- **field** (`django.db.models.Field`) – The field representing the column to change.
- **old\_value** (`int`, *unused*) – The old max length.
- **new\_value** (`int`, *unused*) – The new max length.

**Returns**

The resulting SQL for rebuilding the table.

**Return type**

`django_evolution.db.sql_result.SQLResult`

**change\_column\_type**(*model*, *old\_field*, *new\_field*, *newAttrs*)

Return SQL to change the type of a column.

New in version 2.2.

**Parameters**

- **model** (`type`) – The type of model owning the field.
- **old\_field** (`django.db.models.Field`) – The old field.
- **new\_field** (`django.db.models.Field`) – The new field.
- **newAttrs** (`dict`) – New attributes set in the `ChangeField`.

**Returns**

The SQL statements for changing the column type.

**Return type**

`SQLiteAlterTableSQLResult`

**get\_change\_unique\_sql**(*model*, *field*, *newUniqueValue*, *constraintName*, *initial*)

Change a column's unique flag.

**Parameters**

- **model** (`type`) – The `Model` class representing the table to change the column on.
- **mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation making this change.
- **field** (`django.db.models.Field`) – The field representing the column to change.
- **old\_value** (`bool`, *unused*) – The old unique flag.
- **new\_value** (`bool`, *unused*) – The new unique flag.

**Returns**

The resulting SQL for rebuilding the table.

**Return type**

`django_evolution.db.sql_result.SQLResult`

```
get_update_table_constraints_sql(model, old_constraints, new_constraints, to_add, to_remove)
```

Return SQL for updating the constraints on a table.

This will perform a table rebuild, including only any new constraints in the new schema.

## Parameters



## Returns

The SQL statements for changing the constraints.

## Return type

`django_evolution.sql_result.SQLResult`

**change\_column\_attr\_db\_index**(model, mutation, field, old\_value, new\_value)

Return the SQL for creating/dropping indexes for a column.

If setting `db_index=True`, SQL for generating the index will be returned immediately.

If setting `db_index=False`, the dropping of the index will be scheduled as an Alter Table operation, ensuring it's dropped immediately (and cached/queued database index state updated) before the table is rebuilt, never later.

New in version 2.3.

## Parameters

- **model** (`django.db.models.Model`) – The model being changed.
  - **mutation** (`django_evolution.mutations.BaseModelMutation`) – The mutation applying this change.
  - **field** (`django.db.models.DecimalField`) – The field being modified.
  - **old\_value** (`bool`) – The old value for `db_index`.
  - **new\_value** (`bool`) – The new value for `db_index`.

## Returns

The resulting SQL for creating the index or scheduling a drop.

## Return type

`django_evolution.db.sql_result.SQLResult`

`get_drop_unique_constraint_sql(model, index_name)`

Return SQL for dropping unique constraints.

## Parameters

- **model** (`type`) – The `Model` class representing the table to drop unique constraints on.
- **index\_name** (`unicode`) – The name of the unique constraint index to drop.

**Returns**

The resulting SQL for rebuilding the table.

**Return type**

`django_evolution.db.sql_result.SQLResult`

**get\_indexes\_for\_table(table\_name)**

Return all known indexes on a table.

This is a fallback used only on Django 1.6, due to lack of proper introspection on that release.

**Parameters**

**table\_name** (`unicode`) – The name of the table.

**Returns**

A dictionary mapping index names to a dictionary containing:

**columns (list):**

The list of columns that the index covers.

**unique (bool):**

Whether this is a unique index.

**Return type**

`dict`

**is\_column\_referenced(reffed\_table\_name, reffed\_col\_name)**

Return whether a column on a table is referenced by another table.

**Parameters**

- **reffed\_table\_name** (`unicode`) – The name of the table that may be referenced.
- **reffed\_col\_name** (`unicode`) – The name of the column that may be referenced.

**Returns**

True if this table and column are referenced by another table, or `False` if it's not referenced.

**Return type**

`bool`

**`django_evolution.db.state`**

Database state tracking for in-progress evolutions.

**Classes**

<code>DatabaseState(db_name[, scan])</code>	Tracks some useful state in the database.
<code>IndexState(name[, columns, unique])</code>	An index recorded in the database state.

**`class django_evolution.db.state.IndexState(name, columns=[], unique=False)`**

Bases: `object`

An index recorded in the database state.

### `__init__(name, columns=[], unique=False)`

Initialize the index state.

#### Parameters

- `name` (`unicode`, *optional*) – The name of the index.
- `columns` (`list` of `unicode`, *optional*) – A list of columns that the index is comprised of.
- `unique` (`bool`, *optional*) – Whether this is a unique index.

### `__eq__(other_state)`

Return whether two index states are equal.

#### Parameters

`other_state` (`IndexState`) – The other index state to compare to.

#### Returns

True if the two index states are equal. False if they are not.

#### Return type

`bool`

### `__hash__()`

Return a hash representation of the index.

#### Returns

The hash representation.

#### Return type

`int`

### `__repr__()`

Return a string representation of the index state.

#### Returns

A string representation of the index.

#### Return type

`unicode`

## `class django_evolution.db.state.DatabaseState(db_name, scan=True)`

Bases: `object`

Tracks some useful state in the database.

This primarily tracks indexes associated with tables, allowing them to be scanned from the database, explicitly added, removed, or cleared.

### `__init__(db_name, scan=True)`

Initialize the state.

#### Parameters

- `db_name` (`unicode`) – The name of the database.
- `scan` (`bool`, *optional*) – Whether to automatically scan state from the database during initialization. By default, information is scanned.

### `clone()`

Clone the database state.

#### Returns

The cloned copy of the state.

**Return type**

`DatabaseState`

**`add_table(table_name)`**

Add a table to track.

This will add an empty entry for the table to the state.

**Parameters**

`table_name (unicode)` – The name of the table.

**`has_table(table_name)`**

Return whether a table is being tracked.

This does not necessarily mean that the table exists in the database. Rather, state for the table is being tracked.

**Parameters**

`table_name (unicode)` – The name of the table to look up.

**Returns**

True if the table is being tracked. `False` if it is not.

**Return type**

`bool`

**`has_model(model)`**

Return whether a database model is installed in the database.

**Parameters**

`model (type)` – The model class.

**Returns**

True if the model has an accompanying table in the database. `False` if it does not.

**Return type**

`bool`

**`add_index(table_name, index_name, columns, unique=False)`**

Add a table's index to the database state.

This index can be used for later lookup during the evolution process. It won't otherwise be preserved, though the resulting indexes are expected to match the result in the database.

This requires the table to be tracked first.

**Parameters**

- `table_name (unicode)` – The name of the table.
- `index_name (unicode)` – The name of the index.
- `columns (list of unicode)` – A list of column names the index is comprised of.
- `unique (bool, optional)` – Whether this is a unique index.

**Raises**

`django_evolution.errors.DatabaseStateError` – There was an issue adding this index. Details are in the exception's message.

**`remove_index(table_name, index_name, unique=False)`**

Remove an index from the database state.

This index will no longer be found during lookups when generating evolution SQL, even if it exists in the database.

This requires the table to be tracked first and for the index to both exist and match the unique flag.

### Parameters

- **table\_name** (`unicode`) – The name of the table.
- **index\_name** (`unicode`) – The name of the index.
- **unique** (`bool`, *optional*) – Whether this is a unique index.

### Raises

`djangoevolution.errors.DatabaseStateError` – There was an issue removing this index. Details are in the exception's message.

## `get_index(table_name, index_name, unique=False)`

Return the index state for a given name.

### Parameters

- **table\_name** (`unicode`) – The name of the table.
- **index\_name** (`unicode`) – The name of the index.
- **unique** (`bool`, *optional*) – Whether this is a unique index.

New in version 2.2.

### Returns

The state for the index, if found. `None` if the index could not be found.

### Return type

`IndexState`

## `find_index(table_name, columns, unique=False)`

Find and return an index matching the given columns and flags.

### Parameters

- **table\_name** (`unicode`) – The name of the table.
- **columns** (`list` of `unicode`) – The list of columns the index is comprised of.
- **unique** (`bool`, *optional*) – Whether this is a unique index.

### Returns

The state for the index, if found. `None` if an index matching the criteria could not be found.

### Return type

`IndexState`

## `clear_indexes(table_name)`

Clear all recorded indexes for a table.

### Parameters

`table_name` (`unicode`) – The name of the table.

## `iter_indexes(table_name)`

Iterate through all indexes for a table.

### Parameters

`table_name` (`unicode`) – The name of the table.

**Yields**

`IndexState` – An index in the table.

**rescan\_tables()**

Rescan the list of tables from the database.

This will look up all tables found in the database, along with information (such as indexes) on those tables.

Existing information on the tables will be flushed.

**django\_evolution.utils.apps**

Utilities for working with apps.

**Functions**

<code>get_app_config_for_app(app)</code>	Return the app configuration for an app.
<code>get_app_label(app)</code>	Return the label of an app.
<code>get_app_name(app)</code>	Return the name of an app.
<code>get_legacy_app_label(app)</code>	Return the label of an app.
<code>import_management_modules()</code>	Import the management modules for all apps.

**django\_evolution.utils.apps.get\_app\_config\_for\_app(app)**

Return the app configuration for an app.

This can only be called if running on Django 1.7 or higher.

**Parameters**

`app` (`module`) – The app's models module to return the configuration for. The models module is used for legacy reasons within Django Evolution.

**Returns**

The app configuration, or `None` if it couldn't be found.

**Return type**

`django.apps.AppConfig`

**django\_evolution.utils.apps.get\_app\_label(app)**

Return the label of an app.

**Parameters**

`app` (`module`) – The app.

**Returns**

The label of the app.

**Return type**

`str`

**django\_evolution.utils.apps.get\_app\_name(app)**

Return the name of an app.

**Parameters**

`app` (`module`) – The app.

**Returns**

The name of the app.

**Return type**

`str`

`django_evolution.utils.apps.get_legacy_app_label(app)`

Return the label of an app.

**Parameters**

`app (module)` – The app.

**Returns**

The label of the app.

**Return type**

`str`

`django_evolution.utils.apps.import_management_modules()`

Import the management modules for all apps.

Management modules often contain signal handlers for pre/post syncdb/migrate events. This will import them correctly for the current version of Django.

**Raises**

`ImportError` – A management module failed to import.

## `django_evolution.utils.datastructures`

Utilities for working with data structures.

New in version 2.1.

### Functions

<code>filter_dup_list_items(items)</code>	Return list items with duplicates filtered out.
<code>merge_dicts(dest, source)</code>	Merge two dictionaries together.

`django_evolution.utils.datastructures.filter_dup_list_items(items)`

Return list items with duplicates filtered out.

The order of items will be preserved, but only the first occurrence of any given item will remain in the list.

New in version 2.1.

**Parameters**

`items (list)` – The list of items.

**Returns**

The resulting de-duplicated list of items.

**Return type**

`list`

`django_evolution.utils.datastructures.merge_dicts(dest, source)`

Merge two dictionaries together.

This will recursively merge a source dictionary into a destination dictionary with the following rules:

- Any keys in the source that aren't in the destination will be placed directly to the destination (using the same instance of the value, not a copy).

- Any lists that are in both the source and destination will be combined by appending the source list to the destination list (and this will not recurse into lists).
- Any dictionaries that are in both the source and destination will be merged using this function.
- Any keys that are not a list or dictionary that exist in both dictionaries will result in a `TypeError`.

New in version 2.1.

#### Parameters

- `dest (dict)` – The destination dictionary to merge into.
- `source (dict)` – The source dictionary to merge into the destination.

#### Raises

`TypeError` – A key was present in both dictionaries with a type that could not be merged.

## `django_evolution.utils.evolutions`

Utilities for working with evolutions and mutations.

### Functions

<code>get_app_mutations(app[, evolution_labels, ...])</code>	Return the mutations on an app provided by the given evolution names.
<code>get_app_pending_mutations(app[, ...])</code>	Return an app's pending mutations provided by the given evolution names.
<code>get_app_upgrade_info(app[, scan_evolutions, ...])</code>	Return the upgrade information to use for a given app.
<code>get_applied_evolutions(app[, database])</code>	Return the list of labels for applied evolutions for a Django app.
<code>get_evolution_app_dependencies(app)</code>	Return dependencies governing all evolutions for an app.
<code>get_evolution_dependencies(app, evolution_label)</code>	Return dependencies for an evolution.
<code>get_evolution_module(app, evolution_label)</code>	Return the module for a given evolution for an app.
<code>get_evolution_sequence(app)</code>	Return the list of evolution labels for a Django app.
<code>get_evolutions_module(app)</code>	Return the evolutions module for an app.
<code>get_evolutions_module_name(app)</code>	Return the name of the evolutions module for an app.
<code>get_evolutions_path(app)</code>	Return the evolutions path for an app.
<code>get_evolutions_source(app)</code>	Return the source for evolutions.
<code>get_unapplied_evolutions(app[, database])</code>	Return the list of labels for unapplied evolutions for a Django app.
<code>has_evolutions_module(app)</code>	Return whether an app has an evolutions module.

### `django_evolution.utils.evolutions.has_evolutions_module(app)`

Return whether an app has an evolutions module.

#### Parameters

`app (module)` – The app module.

#### Returns

True if the app has an evolutions module. False if it does not.

#### Return type

`bool`

### `django_evolution.utils.evolutions.get_evolutions_source(app)`

Return the source for evolutions.

This is used to determine where evolutions are coming from. They can be provided by the app, project, or built into Django Evolution.

#### **Parameters**

`app (module)` – The app module.

#### **Returns**

The evolution source. This is an entry from *EvolutionsSource*.

#### **Return type**

`unicode`

### `django_evolution.utils.evolutions.get_evolutions_module_name(app)`

Return the name of the evolutions module for an app.

New in version 2.1.

#### **Parameters**

`app (module)` – The app.

#### **Returns**

The name of the evolutions module for the app. This is not guaranteed to be importable.

#### **Return type**

`unicode`

### `django_evolution.utils.evolutions.get_evolutions_module(app)`

Return the evolutions module for an app.

#### **Parameters**

`app (module)` – The app.

#### **Returns**

The evolutions module for the app, or `None` if it could not be found.

#### **Return type**

`module`

### `django_evolution.utils.evolutions.get_evolution_module(app, evolution_label)`

Return the module for a given evolution for an app.

New in version 2.1.

#### **Parameters**

- `app (module)` – The app.
- `evolution_label (unicode)` – The label of the evolution.

#### **Returns**

The evolution module, or `None` if it could not be found.

#### **Return type**

`module`

### `django_evolution.utils.evolutions.get_evolutions_path(app)`

Return the evolutions path for an app.

#### **Parameters**

`app (module)` – The app.

**Returns**

The path to the evolutions module for the app, or `None` if it could not be found.

**Return type**

`str`

`django_evolution.utils.evolutions.get_evolution_sequence(app)`

Return the list of evolution labels for a Django app.

**Parameters**

`app` (`module`) – The app to return evolutions for.

**Returns**

The list of evolution labels.

**Return type**

`list` of `unicode`

`django_evolution.utils.evolutions.get_evolution_dependencies(app, evolution_label, custom_evolutions=[])`

Return dependencies for an evolution.

Evolutions can depend on other evolutions or migrations, and can be marked as being a dependency of them as well (forcing the evolution to apply before another evolution/migration).

Dependencies are generally specified as a tuple of `(app_label, name)`, where `name` is either a migration name or an evolution label.

Dependencies on evolutions can also be specified as simply a string containing an app label, which will reference the sequence of evolutions as a whole for that app.

Changed in version 2.2: Added the `custom_evolutions` argument.

New in version 2.1.

**Parameters**

- `app` (`module`) – The app the evolution is for.
- `evolution_label` (`unicode`) – The label identifying the evolution for the app.
- `custom_evolutions` (`list` of `dict`, *optional*) – An optional list of custom evolutions pertaining to the app, which will be searched if a module for `evolution_label` could not be found.

Each item is a dictionary containing:

**Keys**

- `label` (`unicode`) – The evolution label (which `evolution_label` will be compared against).
- `after_evolutions` (`list`, *optional*) – A list of evolutions that this would apply after. Each item can be a string (indicating an evolution label within this app) or a tuple in the form of:  
`('app_label', 'evolution_label')`
- `after_migrations` (`list` of `tuple`, *optional*) – A list of migrations that this would apply after. Each item must be a tuple in the form of:  
`('app_label', 'migration_name')`

- **after\_evolutions** (`list`, *optional*) – A list of evolutions that this would apply before. Each item can be a string (indicating an evolution label within this app) or a tuple in the form of:

(‘app\_label’, ‘evolution\_label’)

- **after\_migrations** (`list of tuple`, *optional*) – A list of migrations that this would apply before. Each item must be a tuple in the form of:

(‘app\_label’, ‘migration\_name’)

New in version 2.2.

### Returns

A dictionary of dependency information for the evolution. This has the following keys:

- `before_migrations`
- `after_migrations`
- `before_evolutions`
- `after_evolutions`

If the evolution module was not found, this will return `None` instead.

### Return type

`dict`

`django_evolution.utils.evolutions.get_evolution_app_dependencies(app)`

Return dependencies governing all evolutions for an app.

These dependencies are defined in an `evolutions/__init__.py` file, and will ensure that other evolutions or migrations apply either before or after the app’s evolutions.

Dependencies are generally specified as a tuple of `(app_label, name)`, where `name` is either a migration name or an evolution label.

Dependencies on evolutions can also be specified as simply a string containing an app label, which will reference the sequence of evolutions as a whole for that app.

New in version 2.1.

### Parameters

`app` (`module`) – The app the evolution is for.

### Returns

A dictionary of dependency information for the app. This has the following keys:

- `before_migrations`
- `after_migrations`
- `before_evolutions`
- `after_evolutions`

If the evolutions module was not found, this will return `None` instead.

### Return type

`dict`

`django_evolution.utils.evolutions.get_unapplied_evolutions(app, database='default')`

Return the list of labels for unapplied evolutions for a Django app.

### Parameters

- **app** (`module`) – The app to return evolutions for.
- **database** (`unicode`, *optional*) – The name of the database containing the *Evolution* entries.

**Returns**

The labels of evolutions that have not yet been applied.

**Return type**

`list of unicode`

`django_evolution.utils.evolutions.get_applied_evolutions(app, database='default')`

Return the list of labels for applied evolutions for a Django app.

**Parameters**

- **app** (`module`) – The app to return evolutions for.
- **database** (`unicode`, *optional*) – The name of the database containing the *Evolution* entries.

**Returns**

The labels of evolutions that have been applied.

**Return type**

`list of unicode`

`django_evolution.utils.evolutions.get_app_mutations(app, evolution_labels=None, database='default')`

Return the mutations on an app provided by the given evolution names.

**Parameters**

- **app** (`module`) – The app the evolutions belong to.
- **evolution\_labels** (`list of unicode`, *optional*) – The labels of the evolutions to return mutations for.  
If `None`, this will factor in all evolution labels for the app.
- **database** (`unicode`, *optional*) – The name of the database the evolutions cover.

**Returns**

The list of mutations provided by the evolutions.

**Return type**

`list of django_evolution.mutations.BaseMutation`

**Raises**

`django_evolution.errors.EvolutionException` – One or more evolutions are missing.

`django_evolution.utils.evolutions.get_app_pending_mutations(app, evolution_labels=[], mutations=None, old_project_sig=None, project_sig=None, database='default')`

Return an app's pending mutations provided by the given evolution names.

This is similar to `get_app_mutations()`, but filters the list of mutations down to remove any that are unnecessary (ones that do not operate on changed parts of the project signature).

**Parameters**

- **app** (`module`) – The app the evolutions belong to.

- **evolution\_labels** (`list` of `unicode`, *optional*) – The labels of the evolutions to return mutations for.  
If None, this will factor in all evolution labels for the app.
- **mutations** (`list` of `django_evolution.mutations.BaseMutation`, *optional*) – An explicit list of mutations to use. If provided, `evolution_labels` will be ignored.
- **old\_project\_sig** (`django_evolution.signature.ProjectSignature`, *optional*) – A pre-fetched old project signature. If provided, this will be used instead of the latest one in the database.
- **project\_sig** (`django_evolution.signature.ProjectSignature`, *optional*) – A project signature representing the current state of the database. If provided, this will be used instead of generating a new one from the current database state.
- **database** (`unicode`, *optional*) – The name of the database the evolutions cover.

### Returns

The list of mutations provided by the evolutions.

### Return type

`list` of `django_evolution.mutations.BaseMutation`

### Raises

`django_evolution.errors.EvolutionException` – One or more evolutions are missing.

`django_evolution.utils.evolutions.get_app_upgrade_info(app, scan_evolutions=True, simulate_applied=False, database=None)`

Return the upgrade information to use for a given app.

This will determine if the app should be using Django Evolution or Django Migrations for any schema upgrades.

If an `evolutions` module is found, then this will determine the method to be `UpgradeMethod.EVOLUTIONS`, unless the app has been moved over to using Migrations.

If instead there's a `migrations` module, then this will determine the method to be `UpgradeMethod.MIGRATIONS`.

Otherwise, this will return `None`, indicating that no established method has been chosen. This allows a determination to be made later, based on the Django version or the consumer's choice.

Note that this may return that migrations are the preferred method for an app even on versions of Django that do not support migrations. It's up to the caller to handle this however it chooses.

### Parameters

- **app** (`module`) – The app module to determine the upgrade method for.
- **scan\_evolutions** (`bool`, *optional*) – Whether to scan evolutions for the app to determine the current upgrade method.
- **simulate\_applied** (`bool`, *optional*) – Return the upgrade method based on the state of the app if all mutations had been applied. This is useful for generating end state signatures.

This is ignored if passing `scan_evolutions=False`.

- **database** (`unicode`, *optional*) – The database to use for accessing stored evolution and migration information.

### Returns

A dictionary of information containing the following keys:

**applied\_migrations (*MigrationList*):**

A list of migrations that have been applied to this app through any mutations. This will only be present if the upgrade method is set to use migrations and if running on a version of Django that supports migrations.

**has\_evolutions (*bool*):**

Whether there are any evolutions for this app. This may come from the app, project, or Django Evolution.

**has\_migrations (*bool*):**

Whether there are any migrations for this app.

**upgrade\_method (*unicode*):**

The upgrade method. This will be a value from *UpgradeMethod*, or *None* if a clear determination could not be made.

**Return type**

*dict*

**django\_evolution.utils.graph**

Dependency graphs for tracking and ordering evolutions and migrations.

New in version 2.1.

**Classes**

<i>DependencyGraph()</i>	A graph tracking dependencies between nodes.
<i>EvolutionGraph(*args, **kwargs)</i>	A graph tracking dependencies between migrations and evolutions.
<i>Node(key, insert_index, state)</i>	A node in a graph.

**Exceptions**

<i>NodeNotFoundError(key)</i>	A requested node could not be found.
-------------------------------	--------------------------------------

**exception django\_evolution.utils.graph.NodeNotFoundError(*key*)**

Bases: *Exception*

A requested node could not be found.

New in version 2.1.

**\_\_init\_\_ (*key*)**

Initialize the error.

**Parameters**

**key (*unicode*)** – The key corresponding to the missing node.

**class django\_evolution.utils.graph.Node(*key, insert\_index, state*)**

Bases: *object*

A node in a graph.

Each node is associated with a key, and tracks caller-provided state, dependency relations (in both directions), and an insertion order (for loose sorting).

New in version 2.1.

### **dependencies**

Any other nodes that this node depends on.

#### Type

`set of Node`

### **insert\_index**

An index defining when this was added to the graph, relative to other nodes.

#### Type

`int`

### **key**

The key identifying this node.

#### Type

`unicode`

### **required\_by**

Any other nodes that have this node as a dependency.

#### Type

`set of Node`

### **state**

Tracked state provided by the caller.

#### Type

`dict`

### **\_\_init\_\_(key, insert\_index, state)**

Initialize the node.

#### Parameters

- **key** (`unicode`) – The key identifying this node.
- **insert\_index** (`int`) – An index defining when this was added to the graph, relative to other nodes.
- **state** (`dict`) – Tracked state provided by the caller.

### **\_\_hash\_\_()**

Return a hash of this node.

The hash will be based on the key.

#### Returns

The hash for this node.

#### Return type

`int`

### **\_\_repr\_\_()**

Return a string representation of this node.

#### Returns

The string representation.

---

**Return type**  
`unicode`

**class** `django_evolution.utils.graph.DependencyGraph`

Bases: `object`

A graph tracking dependencies between nodes.

This is used to model relations between objects, indicating which nodes require which, or are required by others, and then providing a sorted order based on those relations.

Dependencies can be added at any time, and are only applied once the graph is finalized. This allows nodes to be added after a dependency referring to them is added.

New in version 2.1.

**`__init__()`**  
 Initialize the graph.

**`add_node(key, state={})`**  
 Add a node to the graph.

A node can only be added if the graph has not been finalized and if the key has not already been recorded.

**Parameters**

- **key** (`unicode`) – The key uniquely identifying this node.
- **state** (`dict, optional`) – State to add to the node.

**Returns**  
 The resulting node.

**Return type**  
`Node`

**`add_dependency(node_key, dep_node_key)`**  
 Add a dependency between two nodes.

This will be recorded as a pending dependency and later applied to the nodes when calling `finalize()`.

**Parameters**

- **node\_key** (`unicode`) – The key of the node that depends on another node.
- **dep\_node\_key** (`unicode`) – The key of the node that `node_key` depends on.

**`remove_dependencies(node_keys)`**  
 Remove any pending dependencies referencing one or more keys.

**Parameters**

**`node_keys`** (`set`) – A set of node keys that should be removed from pending dependencies.

**`finalize()`**  
 Finalize the graph.

This will apply any dependencies and then mark the graph as finalized. At this point, orders can be computed, but no new nodes or dependencies can be added.

**`get_node(key)`**  
 Return a node with a corresponding key.

### Parameters

**key** (`unicode`) – The key associated with the node.

### Returns

The resulting node.

### Return type

`Node`

### Raises

`NodeNotFoundError` – The node could not be found.

## `get_leaf_nodes()`

Return all leaf nodes on the graph.

Leaf nodes are nodes that nothing depends on. These are generally the last evolutions/migrations in any branch of the tree to apply.

### Returns

The list of leaf nodes, sorted by their insertion index.

### Return type

`list` of `Node`

## `get_ordered()`

Return all nodes in dependency order.

This will perform a topological sort on the graph, returning nodes in the order they should be processed in.

The graph must be finalized before this is called.

### Returns

The list of nodes, in dependency order.

### Return type

`list` of `Node`

## `class django_evolution.utils.graph.EvolutionGraph(*args, **kwargs)`

Bases: `DependencyGraph`

A graph tracking dependencies between migrations and evolutions.

This is used to model the relationships between all configured migrations and evolutions, and to generate batches of consecutive migrations or evolutions that can be applied at once.

Dependencies can be added at any time, and are only applied once the graph is finalized. This allows nodes to be added after a dependency referring to them is added.

New in version 2.1.

### `NODE_TYPE_ANCHOR = 'anchor'`

An anchor node.

These are internal, and are used for clustered dependency management.

### `NODE_TYPE_CREATE_MODEL = 'create-model'`

A node that results in model creation.

### `NODE_TYPE_EVOLUTION = 'evolution'`

A node that results in applying a single evolution.

---

```
NODE_TYPE_MIGRATION = 'migration'
```

A node that results in applying a single migration.

```
__init__(*args, **kwargs)
```

Initialize the graph.

#### Parameters

- **\*args** (`tuple`) – Positional arguments for the parent.
- **\*\*kwargs** (`dict`) – Keyword arguments for the parent.

```
add_evolutions(app, evolutions=[], new_models=[], extra_state={}, custom_evolutions=[])
```

Add a list of evolutions for a given app.

Each evolution will get its own node, and pending dependencies will be recorded to ensure the evolutions are applied in the correct order.

A special `__first__` anchor node will be added before the sequence of evolutions, and a `__last__` node will be added after. This allows evolutions to easily reference another app's list of evolutions relative to the start or end of a list. It's used only internally.

Changed in version 2.2: A `custom_evolutions` argument can now be provided, for dependency resolution purposes.

#### Parameters

- **app** (`module`) – The app module the evolutions apply to.
- **evolutions** (`list` of `django_evolution.models.Evolution`, *optional*) – The list of evolutions to add to the graph. This may be an empty list if there are no evolutions but there are new models to create.
- **new\_models** (`list` of `type`, *optional*) – The list of database model classes to create for the app.
- **extra\_state** (`dict`, *optional*) – Extra state to set in each evolution node.
- **custom\_evolutions** (`list` of `dict`, *optional*) – An optional list of custom evolutions for the app, for dependency resolution.

New in version 2.2.

```
add_migration_plan(migration_plan, migration_graph)
```

Add a migration plan to the graph.

Each migration in the plan will get its own node, and pending dependencies will be recorded to ensure the migrations are applied in the order already computed for the plan.

#### Parameters

- **migration\_plan** (`list` of `tuple`) – The computed migration plan to add to the graph.
- **migration\_graph** (`django.db.migrations.graph.MigrationGraph`) – The computed migration graph, used to reference computed dependencies.

```
mark_evolutions_applied(app, evolution_labels)
```

Mark one or more evolutions as applied.

This will remove any pending dependencies referencing these evolutions from the graph.

#### Parameters

- **app** (`module`) – The app module the evolutions apply to.

- **evolution\_labels** (`list of unicode`) – The list of evolutions labels to mark as applied.

### `mark_migrations_applied(migrations)`

Mark one or more migrations as applied.

This will remove any pending dependencies referencing these migrations from the graph.

#### Parameters

**migrations** (`django_evolution.utils.migrations.MigrationList`) – The list of migrations to mark as applied.

### `iter_batches()`

Iterate through batches of consecutive evolutions and migrations.

The nodes will be iterated in dependency order, with each batch containing a sequence of either evolutions or migrations that can be applied at once.

#### Yields

`tuple` – A 2-tuple containing:

1. The batch type (one of `NODE_TYPE_CREATE_MODEL`, `NODE_TYPE_EVOLUTION`, or `NODE_TYPE_MIGRATION`).
2. A list of `Node` instances.

## `django_evolution.utils.migrations`

Utility functions for working with Django Migrations.

## Functions

<code>apply_migrations(executor, targets, plan, ...)</code>	Apply migrations to the database.
<code>clear_global_custom_migrations()</code>	Clear the list of custom migrations.
<code>create_pre_migrate_state(executor)</code>	Create state needed before migrations are applied.
<code>emit_post_migrate_or_sync(verbosity, ...)</code>	Emit the post_migrate and/or post_sync signals.
<code>emit_pre_migrate_or_sync(verbosity, ...)</code>	Emit the pre_migrate and/or pre_sync signals.
<code>filter_migration_targets(targets[, ...])</code>	Filter migration execution targets based on the given criteria.
<code>finalize_migrations(post_migrate_state)</code>	Finalize any migrations operations.
<code>has_migrations_module(app)</code>	Return whether an app has a migrations module.
<code>is_migration_initial(migration)</code>	Return whether a migration is an initial migration.
<code>record_applied_migrations(connection, migrations)</code>	Record a list of applied migrations to the database.
<code>register_global_custom_migrations(...)</code>	Register a global list of custom migrations.
<code>unrecord_applied_migrations(connection, ...)</code>	Remove the recordings of applied migrations from the database.

## Classes

<code>MigrationExecutor(connection[, ...])</code>	Load and execute migrations.
<code>MigrationList()</code>	A list of applied or pending migrations.
<code>MigrationLoader(connection[, custom_migrations])</code>	Loads migration files from disk.

`class django_evolution.utils.migrations.MigrationList`

Bases: `object`

A list of applied or pending migrations.

This is used to manage a list of migrations in a way that's independent from the underlying representation used in Django. Migrations are tracked by app label and name, may be associated with a recorded migration database entry, and can be used to convert state to and from both signatures and Django migration state.

`classmethod from_app_sig(app_sig)`

Create a `MigrationList` based on an app signature.

**Parameters**

`app_sig (django_evolution.signature.AppSignature)` – The app signature containing a list of applied migrations.

**Returns**

The new migration list.

**Return type**

`MigrationList`

`classmethod from_names(app_label, migration_names)`

Create a `MigrationList` based on a list of migration names.

New in version 2.1.

**Parameters**

- `app_label (unicode)` – The app label common to each migration name.
- `migration_names (list of unicode)` – The list of migration names.

**Returns**

The new migration list.

**Return type**

`MigrationList`

`classmethod from_database(connection, app_label=None)`

Create a `MigrationList` based on recorded migrations.

**Parameters**

- `connection (django.db.backends.base.BaseDatabaseWrapper)` – The database connection used to query for migrations.
- `app_label (unicode, optional)` – An app label to filter migrations by.

**Returns**

The new migration list.

**Return type**

`MigrationList`

### `__init__()`

Initialize the list.

### `has_migration_info(app_label, name)`

Return whether the list contains an entry for a migration.

#### Parameters

- `app_label` (`unicode`) – The label for the application that was migrated.
- `name` (`unicode`) – The name of the migration.

#### Returns

True if the migration is in the list. `False` if it is not.

#### Return type

`bool`

### `add_migration_targets(targets)`

Add a list of migration targets to the list.

#### Parameters

- `targets` (`list` of `tuple`) – The migration targets to each. Each is a tuple containing an app label and a migration name.

### `add_migration(migration)`

Add a migration to the list.

This can only be called on Django 1.7 or higher.

#### Parameters

- `migration` (`django.db.migrations.Migration`) – The migration instance to add.

### `add_recorded_migration(recorded_migration)`

Add a recorded migration to the list.

This can only be called on Django 1.7 or higher.

#### Parameters

- `recorded_migration` (`django.db.migrations.recorder.MigrationRecorder.Migration`) – The recorded migration model to add.

### `add_migration_info(app_label, name, migration=None, recorded_migration=None)`

Add information on a migration to the list.

#### Parameters

- `app_label` (`unicode`) – The label for the application that was migrated.
- `name` (`unicode`) – The name of the migration.
- `migration` (`django.db.migrations.Migration`, *optional*) – An optional migration instance to associate with this entry.
- `recorded_migration` (`django.db.migrations.recorder.MigrationRecorder.Migration`, *optional*) – An optional recorded migration to associate with this entry.

### `update(other)`

Update the list with the contents of another list.

If there's an entry in another list matching this one, and contains information that the entry in this list does not have, this list's entry will be updated.

**Parameters**

**other** (*MigrationList*) – The list of migrations to put into this list.

**to\_targets()**

Return a set of migration targets based on this list.

**Returns**

A set of migration targets. Each entry is a tuple containing the app label and name.

**Return type**

`set`

**get\_app\_labels()**

Iterate through the app labels.

Results are sorted alphabetically.

**Returns**

The sorted list of app labels with associated migrations.

**Return type**

`list` of `unicode`

**clone()**

Clone the list.

**Returns**

The cloned migration list.

**Return type**

*MigrationList*

**\_\_bool\_\_()**

Return whether this list is truthy or falsy.

The list is truthy only if it has items.

**Returns**

True if the list has items. `False` if it's empty.

**Return type**

`bool`

**\_\_len\_\_()**

Return the number of items in the list.

**Returns**

The number of items in the list.

**Return type**

`int`

**\_\_eq\_\_(other)**

Return whether this list is equal to another list.

The order of migrations is ignored when comparing lists.

**Parameters**

**other** (*MigrationList*) – A list of migrations to compare to.

**Returns**

True if the two lists have the same contents. `False` if there are differences in contents, or `other` is not a *MigrationList*.

**Return type**

`bool`

**`__iter__()`**

Iterate through the list.

Entries are sorted first by app label, alphabetically, and then the order in which migrations were added for that app label.

**Yields**

`info` – A dictionary containing the following keys:

**`app_label (unicode):`**

The app label for the migration.

**`name (unicode):`**

The name of the migration.

**`migration (django.db.migrations.Migration):`**

The optional migration instance.

**`recorded_migration`****`(django.db.migrations.recorder.MigrationRecorder.Migration):`**

The optional recorded migration.

**`__add__(other)`**

Return a combined copy of this list and another list.

**Parameters**

`other (MigrationList)` – The other list to add to this list.

**Returns**

The new migration list containing contents of both lists.

**Return type**

`MigrationList`

**`__sub__(other)`**

Return a copy of this list with another list's contents excluded.

**Parameters**

`other (MigrationList)` – The other list containing contents to exclude.

**Returns**

The new migration list containing the contents of this list that don't exist in the other list.

**Return type**

`MigrationList`

**`__repr__()`**

Return a string representation of this list.

**Returns**

The string representation.

**Return type**

`unicode`

**`__hash__ = None`**

---

```
class django_evolution.utils.migrations.MigrationLoader(connection, custom_migrations=None,
                                                       *args, **kwargs)
```

Bases: `MigrationLoader`

Loads migration files from disk.

This is a specialization of Django's own `MigrationLoader` that allows for providing additional migrations not available on disk.

#### `extra_applied_migrations`

Migrations to mark as already applied. This can be used to augment the results calculated from the database.

##### Type

`MigrationList`

```
__init__(connection, custom_migrations=None, *args, **kwargs)
```

Initialize the loader.

##### Parameters

- `connection` (`django.db.backends.base.BaseDatabaseWrapper`) – The connection to load applied migrations from.
- `custom_migrations` (`MigrationList`, optional) – Custom migrations not available on disk.
- `*args` (`tuple`) – Additional positional arguments for the parent class.
- `**kwargs` (`dict`) – Additional keyword arguments for the parent class.

#### `property applied_migrations`

The migrations already applied.

This will contain both the migrations applied from the database and any set in `extra_applied_migrations`.

```
build_graph(reload_migrations=True)
```

Rebuild the migrations graph.

##### Parameters

`reload_migrations` (`bool`, optional) – Whether to reload migration instances from disk. If `False`, the ones loaded before will be used.

#### `load_disk()`

Load migrations from disk.

This will also load any custom migrations.

---

```
class django_evolution.utils.migrations.MigrationExecutor(connection, custom_migrations=None,
                                                       signal_sender=None)
```

Bases: `MigrationExecutor`

Load and execute migrations.

This is a specialization of Django's own `MigrationExecutor` that allows for providing additional migrations not available on disk, and for emitting our own signals when processing migrations.

```
__init__(connection, custom_migrations=None, signal_sender=None)
```

Initialize the executor.

Changed in version 2.2: `custom_migrations` now defaults to any globally-registered custom migrations set in `register_global_custom_migrations()`.

### Parameters

- **connection** (`django.db.backends.base.BaseDatabaseWrapper`) – The connection to load applied migrations from.
- **custom\_migrations** (`dict`, *optional*) – Custom migrations not available on disk. Each key is a tuple of (`app_label`, `migration_name`), and each value is a migration.  
This defaults to any globally-registered custom migrations.
- **signal\_sender** (`object`, *optional*) – A custom sender to pass when sending signals. This defaults to this instance.

### `run_checks()`

Perform checks on the migrations and any history.

### Raises

- `django_evolution.errors.MigrationConflictsError` – There are conflicts between migrations loaded from disk.
- `django_evolution.errors.MigrationHistoryError` – There are unapplied dependencies to applied migrations.

### `django_evolution.utils.migrations.register_global_custom_migrations(custom_migrations)`

Register a global list of custom migrations.

These will be used by default when constructing a `MigrationExecutor`.

Only one list of custom migrations can be added at a time.

This is primarily useful for unit testing.

New in version 2.2.

### Parameters

`custom_migrations` (`MigrationList`) – The list of custom migrations.

### Raises

`AssertionError` – Custom migrations were already registered.

### `django_evolution.utils.migrations.clear_global_custom_migrations()`

Clear the list of custom migrations.

New in version 2.2.

### `django_evolution.utils.migrations.has_migrations_module(app)`

Return whether an app has a migrations module.

### Parameters

`app` (`module`) – The app module.

### Returns

True if the app has a `migrations` module. False if it does not.

### Return type

`bool`

### `django_evolution.utils.migrations.record_applied_migrations(connection, migrations)`

Record a list of applied migrations to the database.

This can only be called when on Django 1.7 or higher.

### Parameters

- **connection** (`django.db.backends.base.BaseDatabaseWrapper`) – The connection used to record applied migrations.
- **migrations** (`MigrationList`) – The list of migration targets to record as applied.

```
django_evolution.utils.migrations.unrecord_applied_migrations(connection, app_label, migration_names=None)
```

Remove the recordings of applied migrations from the database.

This can only be called when on Django 1.7 or higher.

#### Parameters

- **connection** (`django.db.backends.base.BaseDatabaseWrapper`) – The connection used to unrecord applied migrations.
- **app\_label** (`unicode`) – The app label that the migrations pertain to.
- **migration\_names** (`list` of `unicode`, *optional*) – The list of migration names to unrecord. If not provided, all migrations for the app will be unrecorded.

```
django_evolution.utils.migrations.filter_migration_targets(targets, app_labels=None, exclude=None)
```

Filter migration execution targets based on the given criteria.

#### Parameters

- **targets** (`list` of `tuple`) – The migration targets to be executed.
- **app\_labels** (`set` of `unicode`, *optional*) – The app labels to limit the targets to.
- **exclude** (`set`, *optional*) – Explicit targets to exclude.

#### Returns

The resulting list of migration targets.

#### Return type

`list` of `tuple`

```
django_evolution.utils.migrations.is_migration_initial(migration)
```

Return whether a migration is an initial migration.

Initial migrations are those that set up an app or models for the first time. Generally, they should be limited to model creations, or to those adding fields to a (non-migration-aware) model for the first time. They also should not have any dependencies on other migrations within the same app.

An initial migration should be able to be safely soft-applied (in other words, ignored if the model already appears to exist in the database).

Migrations on Django 1.9+ may declare themselves as explicitly initial or explicitly not initial.

#### Parameters

**migration** (`django.db.migrations.Migration`) – The migration to check.

#### Returns

True if the migration appears to be an initial migration. False if it does not.

#### Return type

`bool`

```
django_evolution.utils.migrations.create_pre_migrate_state(executor)
```

Create state needed before migrations are applied.

The return value is dependent on the version of Django.

### Parameters

**executor** (`django.db.migrations.executor.MigrationExecutor`) – The migration executor that will handle the migrations.

### Returns

The state needed for applying migrations.

### Return type

`django.db.migrations.state.ProjectState`

`django_evolution.utils.migrations.apply_migrations(executor, targets, plan, pre_migrate_state)`

Apply migrations to the database.

Migrations will be applied using the `fake_initial` mode, which means that any initial migrations (those constructing the models for an app) will be skipped if the models already appear in the database. This is to avoid issues with applying those migrations when the models have already been created in the past outside of Django's Migrations framework. In theory, this could cause some issues if those migrations also perform other important operations around data population, but this is really up to Django to handle, as this is part of the upgrade method when going from pre-1.7 to 1.7+ anyway.

This can only be called when on Django 1.7 or higher.

### Parameters

- **executor** (`django.db.migrations.executor.MigrationExecutor`) – The migration executor that will handle applying the migrations.
- **targets** (`list of tuple`) – The list of migration targets to apply.
- **plan** (`list of tuple`) – The order in which migrations will be applied.
- **pre\_migrate\_state** (`object`) – The pre-migration state needed to apply these migrations. This must be generated with `create_pre_migrate_state()` or a previous call to `apply_migrations()`.

### Returns

The state generated from applying migrations. Any final state must be passed to `finalize_migrations()`.

### Return type

`object`

`django_evolution.utils.migrations.finalize_migrations(post_migrate_state)`

Finalize any migrations operations.

This will update any internal state in Django for any migrations that were applied and represented by the provided post-migrate state.

### Parameters

**post\_migrate\_state** (`object`) – The state generated from applying migrations. This must be the result of `apply_migrations()`.

`django_evolution.utils.migrations.emit_pre_migrate_or_sync(verbosity, interactive, database_name, create_models, pre_migrate_state, plan)`

Emit the pre\_migrate and/or pre\_sync signals.

This will emit the `pre_migrate` and/or `pre_sync` signals, providing the appropriate arguments for the current version of Django.

### Parameters

- **verbosity** (`int`) – The verbosity level for output.

- **interactive** (`bool`) – Whether handlers of the signal can prompt on the terminal for input.
- **database\_name** (`unicode`) – The name of the database being migrated.
- **create\_models** (`list` of `django.db.models.Model`) – The list of models being created outside of any migrations.
- **pre\_migrate\_state** (`django.db.migrations.state.ProjectState`) – The project state prior to any migrations.
- **plan** (`list`) – The full migration plan being applied.

`django_evolution.utils.migrations.emit_post_migrate_or_sync(verbosity, interactive, database_name, created_models, post_migrate_state, plan)`

Emit the post\_migrate and/or post\_sync signals.

This will emit the `post_migrate` and/or `post_sync` signals, providing the appropriate arguments for the current version of Django.

#### Parameters

- **verbosity** (`int`) – The verbosity level for output.
- **interactive** (`bool`) – Whether handlers of the signal can prompt on the terminal for input.
- **database\_name** (`unicode`) – The name of the database that was migrated.
- **created\_models** (`list` of `django.db.models.Model`) – The list of models created outside of any migrations.
- **post\_migrate\_state** (`django.db.migrations.state.ProjectState`) – The project state after applying migrations.
- **plan** (`list`) – The full migration plan that was applied.

## `django_evolution.utils.models`

Utilities for working with models.

### Functions

<code>clear_model_rel_tree()</code>	Clear the model relationship tree.
<code>get_database_for_model_name(app_name, model_name)</code>	Return the database used for a given model.
<code>get_model_rel_tree()</code>	Return the full field relationship tree for all registered models.
<code>iter_model_fields(model[, ...])</code>	Iterate through all fields on a model using the given criteria.
<code>iter_non_m2m_reverse_relations(field)</code>	Iterate through non-M2M reverse relations pointing to a field.
<code>walk_model_tree(model)</code>	Walk through a tree of models.

### `django_evolution.utils.models.get_database_for_model_name(app_name, model_name)`

Return the database used for a given model.

Given an app name and a model name, this will return the proper database connection name used for making changes to that model. It will go through any custom routers that understand that type of model.

#### Parameters

- `app_name` (`unicode`) – The name of the app owning the model.
- `model_name` (`unicode`) – The name of the model.

#### Returns

The name of the database used for the model.

#### Return type

`unicode`

### `django_evolution.utils.models.walk_model_tree(model)`

Walk through a tree of models.

This will yield the provided model and its parents, in turn yielding their parents, and so on.

New in version 2.2.

#### Parameters

- `model` (`type`) – The top of the model tree to iterate through.

#### Yields

`type` – Each model class in the tree.

### `django_evolution.utils.models.get_model_rel_tree()`

Return the full field relationship tree for all registered models.

This will walk through every field in every model registered in Django, storing the relationships between objects, caching them. Each entry in the resulting dictionary will be a table mapping to a list of relation fields that point back at it.

This can be used to quickly locate any and all reverse relations made to a field.

This is similar to Django's built-in reverse relation tree used internally (with different implementations) in `django.db.models.options.Options`, but works across all supported versions of Django, and supports cache clearing.

New in version 2.2.

#### Returns

The model relation tree.

#### Return type

`dict`

### `djangoevolution.utils.models.clear_model_rel_tree()`

Clear the model relationship tree.

This will cause the next call to `get_model_rel_tree()` to re-compute the full tree.

New in version 2.2.

### `django_evolution.utils.models.iter_model_fields(model, include_parent_models=True, include_forward_fields=True, include_reverse_fields=False, include_hidden_fields=False, seen_models=None)`

Iterate through all fields on a model using the given criteria.

This is roughly equivalent to Django's internal `django.db.models.options.Option._get_fields()` on Django 1.8+, but makes use of our model reverse relation tree, and works across all supported versions of Django.

New in version 2.2.

#### Parameters

- `model` (`type`) – The model owning the fields.
- `include_parent_models` (`bool`, *optional*) – Whether to include fields defined on parent models.
- `include_forward_fields` (`bool`, *optional*) – Whether to include fields owned by the model (or a parent).
- `include_reverse_fields` (`bool`, *optional*) – Whether to include fields on other models that point to this model.
- `include_hidden_fields` (`bool`, *optional*) – Whether to include hidden fields.
- `seen_models` (`set`, *optional*) – Models seen during iteration. This is intended for internal use only by this function.

#### Yields

`django.db.models.Field` – Each field matching the criteria.

### `django_evolution.utils.models.iter_non_m2m_reverse_relations(field)`

Iterate through non-M2M reverse relations pointing to a field.

This will exclude any :py:class:`~django.db.models.ManyToManyField`'s, but will include the relation fields on their "through" tables.

Note that this may return duplicate results, or multiple relations pointing to the same field. It's up to the caller to handle this.

New in version 2.2.

#### Parameters

- `field` (`django.db.models.Field`) – The field that relations must point to.

#### Yields

`django.db.models.Field` or `object` – Each field or relation object pointing to this field.

The type of the relation object depends on the version of Django.

## `django_evolution.utils.sql`

Utilities for working with SQL statements.

### Classes

<code>BaseGroupedSQL(sql)</code>	Base class for a grouped list of SQL statements.
<code>NewTransactionSQL(sql)</code>	A list of SQL statements to execute in its own transaction.
<code>NoTransactionSQL(sql)</code>	A list of SQL statements to execute outside of a transaction.
<code>SQLExecutor(database[, check_constraints])</code>	Management for the execution of SQL.

```
class django_evolution.utils.sql.BaseGroupedSQL(sql)
```

Bases: `object`

Base class for a grouped list of SQL statements.

This is a simple wrapper around a list of SQL statements, used to group statements under some category defined by a subclass.

### `sql`

A list of SQL statements, as allowed by `run_sql()`.

#### Type

`list`

### `__init__(sql)`

Initialize the group.

#### Parameters

`sql (list)` – A list of SQL statements, as allowed by `run_sql()`.

```
class django_evolution.utils.sql.NewTransactionSQL(sql)
```

Bases: `BaseGroupedSQL`

A list of SQL statements to execute in its own transaction.

```
class django_evolution.utils.sql.NoTransactionSQL(sql)
```

Bases: `BaseGroupedSQL`

A list of SQL statements to execute outside of a transaction.

```
class django_evolution.utils.sql.SQLExecutor(database, check_constraints=True)
```

Bases: `object`

Management for the execution of SQL.

This allows callers to perform raw SQL queries against the database, and to do so with a fine degree of transaction management. Callers can continually add new SQL to execute and, in-between, enter into a new transaction, ensure a previous transaction is already open, or close out any existing transaction.

Through this, it can effectively script a set of transactions and queries in a more loose form than normally allowed by Django.

New in version 2.1.

### `__init__(database, check_constraints=True)`

Initialize the executor.

#### Parameters

- `database (unicode)` – The registered database name where queries will be executed.
- `check_constraints (bool, optional)` – Whether to check constraints during the execution of SQL. If disabled, it's up to the caller to manually invoke a constraint check.

### `__enter__()`

Enter the context manager.

This will prepare internal state for execution, and optionally disable constraint checking (if requested during construction).

The context manager must be entered before operations will work.

**Context**

`SQLExecutor` – This instance.

**`__exit__(*args, **kwargs)`**

Exit the context manager.

This will commit any transaction that may be in progress, close the database cursor, and re-enable constraint checking if it were previously disabled.

**Parameters**

- `*args (tuple, unused)` – Unused positional arguments.
- `**kwargs (dict, unused)` – Unused keyword arguments.

**`new_transaction()`**

Start a new transaction.

This will commit any prior transaction, if one exists, and then start a new one.

**`ensure_transaction()`**

Ensure a transaction has started.

If no existing transaction has started, this will start a new one.

**`finish_transaction()`**

Finish and commit a transaction.

**`run_sql(sql, capture=False, execute=False)`**

Run (execute and/or capture) a list of SQL statements.

**Parameters**

- `sql (list)` – A list of SQL statements. Each entry might be a string, a tuple consisting of a format string and formatting arguments, or a subclass of `BaseGroupedSQL`, or a callable that returns a list of the above.
- `capture (bool, optional)` – Whether to capture any processed SQL statements.
- `execute (bool, optional)` – Whether to execute any executed SQL statements and return them.

**Returns**

The list of SQL statements executed, if passing `capture=True`. Otherwise, this will just be an empty list.

**Return type**

`list of unicode`

**Raises**

`django.db.transaction.TransactionManagementError` – Could not execute a batch of SQL statements inside of an existing transaction.



## PYTHON MODULE INDEX

### d

django\_evolution, 55  
django\_evolution.compat.apps, 147  
django\_evolution.compat.commands, 149  
django\_evolution.compat.datastructures, 150  
django\_evolution.compat.db, 152  
django\_evolution.compat.models, 159  
django\_evolution.compat.picklers, 164  
django\_evolution.compat.py23, 165  
django\_evolution.conf, 56  
django\_evolution.consts, 57  
django\_evolution.db.common, 166  
django\_evolution.db.mysql, 179  
django\_evolution.db.postgresql, 182  
django\_evolution.db.sql\_result, 184  
django\_evolution.db.sqlite3, 186  
django\_evolution.db.state, 191  
django\_evolution.deprecation, 58  
django\_evolution.diff, 133  
django\_evolution.errors, 59  
django\_evolution.evolve, 62  
django\_evolution.evolve.base, 62  
django\_evolution.evolve.evolve\_app\_task, 70  
django\_evolution.evolve.evolver, 65  
django\_evolution.evolve.purge\_app\_task, 73  
django\_evolution.mock\_models, 135  
django\_evolution.models, 74  
django\_evolution.mutations, 78  
django\_evolution.mutations.add\_field, 78  
django\_evolution.mutations.base, 80  
django\_evolution.mutations.change\_field, 87  
django\_evolution.mutations.change\_meta, 88  
django\_evolution.mutations.delete\_application,  
    90  
django\_evolution.mutations.delete\_field, 91  
django\_evolution.mutations.delete\_model, 92  
django\_evolution.mutations.move\_to\_django\_migrations,  
    93  
django\_evolution.mutations.rename\_app\_label,  
    94  
django\_evolution.mutations.rename\_field, 95  
django\_evolution.mutations.rename\_model, 96  
django\_evolution.mutations.sql\_mutation, 97  
django\_evolution.mutators, 139  
django\_evolution.mutators.app\_mutator, 139  
django\_evolution.mutators.model\_mutator, 141  
django\_evolution.mutators.sql\_mutator, 144  
django\_evolution.placeholders, 144  
django\_evolution.serialization, 99  
django\_evolution.signals, 109  
django\_evolution.signature, 110  
django\_evolution.support, 146  
django\_evolution.utils.apps, 195  
django\_evolution.utils.datastructures, 196  
django\_evolution.utils.evolutions, 197  
django\_evolution.utils.graph, 203  
django\_evolution.utils.migrations, 208  
django\_evolution.utils.models, 217  
django\_evolution.utils.sql, 219



# INDEX

## Symbols

\_\_add\_\_(*django\_evolution.utils.migrations.MigrationList*, *method*), 212  
\_\_bool\_\_(*django\_evolution.utils.migrations.MigrationList*, *method*), 211  
\_\_call\_\_(*django\_evolution.placeholders.BasePlaceholder*, *method*), 145  
\_\_call\_\_(*django\_evolution.placeholders.NullFieldInitial*, *method*), 146  
\_\_delitem\_\_(*django\_evolution.compat.datastructures.OrderedDict*, *method*), 150  
\_\_enter\_\_(*django\_evolution.utils.sql.SQLExecutor*, *method*), 220  
\_\_eq\_\_(*django\_evolution.compat.datastructures.OrderedDict*, *method*), 151  
\_\_eq\_\_(*django\_evolution.db.state.IndexState*, *method*), 192  
\_\_eq\_\_(*django\_evolution.mock\_models.MockModel*, *method*), 138  
\_\_eq\_\_(*django\_evolution.mutations.base.BaseMutation*, *method*), 84  
\_\_eq\_\_(*django\_evolution.signature.AppSignature*, *method*), 121  
\_\_eq\_\_(*django\_evolution.signature.BaseSignature*, *method*), 114  
\_\_eq\_\_(*django\_evolution.signature.ConstraintSignature*, *method*), 127  
\_\_eq\_\_(*django\_evolution.signature.FieldSignature*, *method*), 132  
\_\_eq\_\_(*django\_evolution.signature.IndexSignature*, *method*), 129  
\_\_eq\_\_(*django\_evolution.signature.ModelSignature*, *method*), 125  
\_\_eq\_\_(*django\_evolution.signature.ProjectSignature*, *method*), 117  
\_\_eq\_\_(*django\_evolution.utils.migrations.MigrationList*, *method*), 211  
\_\_exit\_\_(*django\_evolution.utils.sql.SQLExecutor*, *method*), 221  
\_\_ge\_\_(*django\_evolution.compat.datastructures.OrderedDict*, *method*), 151  
\_\_get\_\_(*django\_evolution.compat.models.GenericForeignKey*, *method*), 151  
getattr(*django\_evolution.mock\_models.MockMeta*, *method*), 137  
getattribute(*django\_evolution.compat.commands.BaseCommand*, *method*), 150  
gt(*django\_evolution.compat.datastructures.OrderedDict*, *method*), 151  
hash(*django\_evolution.compat.datastructures.OrderedDict*, *attribute*), 151  
hash(*django\_evolution.signature.AppSignature*, *attribute*), 121  
hash(*django\_evolution.signature.BaseSignature*, *attribute*), 115  
hash(*django\_evolution.signature.FieldSignature*, *attribute*), 131  
hash(*django\_evolution.signature.ModelSignature*, *attribute*), 126  
hash(*django\_evolution.signature.ProjectSignature*, *attribute*), 118  
hash(*django\_evolution.utils.migrations.MigrationList*, *attribute*), 212  
hash(*django\_evolution.db.state.IndexState*, *method*), 192  
hash(*django\_evolution.mock\_models.MockModel*, *method*), 138  
hash(*django\_evolution.mutations.base.BaseMutation*, *method*), 84  
hash(*django\_evolution.signature.ConstraintSignature*, *method*), 127  
hash(*django\_evolution.signature.IndexSignature*, *method*), 129  
hash(*django\_evolution.utils.graph.Node*, *method*), 204  
init(*django\_evolution.compat.commands.OptionParserWrapper*, *method*), 149  
init(*django\_evolution.compat.datastructures.OrderedDict*, *method*), 150  
init(*django\_evolution.compat.models.GenericForeignKey*, *method*), 159  
init(*django\_evolution.compat.models.GenericRelation*, *method*), 160  
init(*django\_evolution.conf.DjangoEvolutionSettings*, *method*)

```

        method), 56
__init__(django_evolution.db.common.BaseEvolutionOperationsmethod), 97
        method), 166
__init__(django_evolution.db.sql_result.AlterTableSQLResult
        method), 185
__init__(django_evolution.db.sql_result.SQLResult
        method), 184
__init__(django_evolution.db.state.DatabaseState
        method), 192
__init__(django_evolution.db.state.IndexState
        method), 191
__init__(django_evolution.diff.Diff method), 134
__init__(django_evolution.errors.EvolutionException
        method), 59
__init__(django_evolution.errors.EvolutionExecutionError
        method), 60
__init__(django_evolution.errors.InvalidSignatureVersion
        method), 61
__init__(django_evolution.errors.MigrationConflictsError
        method), 61
__init__(django_evolution.evolve.base.BaseEvolutionTask
        method), 63
__init__(django_evolution.evolve.evolve_app_task.EvolveAppTask
        method), 71
__init__(django_evolution.evolve.evolver.Evolver
        method), 66
__init__(django_evolution.evolve.purge_app_task.PurgeAppTask
        method), 73
__init__(django_evolution.mock_models.MockMeta
        method), 136
__init__(django_evolution.mock_models.MockModel
        method), 138
__init__(django_evolution.mock_models.MockRelated
        method), 139
__init__(django_evolution.mutations.add_field.AddField
        method), 78
__init__(django_evolution.mutations.base.BaseModelFieldMutation
        method), 87
__init__(django_evolution.mutations.base.BaseModelMutation
        method), 86
__init__(django_evolution.mutations.base.Simulation
        method), 80
__init__(django_evolution.mutations.change_field.ChangeField
        method), 87
__init__(django_evolution.mutations.change_meta.ChangeMeta
        method), 89
__init__(django_evolution.mutations.move_to_django_migrations.MoveToDjangoMigrations
        method), 93
__init__(django_evolution.mutations.rename_app_label.RenameAppLabel
        method), 94
__init__(django_evolution.mutations.rename_field.RenameField
        method), 95
__init__(django_evolution.mutations.rename_model.RenameModel
        method), 96
__init__(django_evolution.mutations.sql_mutation.SQLMutation
        method), 97
        method), 140
__init__(django_evolution.mutators.model_mutator.ModelMutator
        method), 141
__init__(django_evolution.mutators.sql_mutator.SQLMutator
        method), 144
__init__(django_evolution.placeholders.BasePlaceholder
        method), 145
__init__(django_evolution.signature.AppSignature
        method), 118
__init__(django_evolution.signature.ConstraintSignature
        method), 126
__init__(django_evolution.signature.FieldSignature
        method), 130
__init__(django_evolution.signature.IndexSignature
        method), 128
__init__(django_evolution.signature.ModelSignature
        method), 122
__init__(django_evolution.signature.ProjectSignature
        method), 115
__init__(django_evolution.utils.graph.DependencyGraph
        method), 205
__init__(django_evolution.utils.graph.EvolutionGraph
        method), 207
__init__(django_evolution.utils.graph.Node
        method), 204
__init__(django_evolution.utils.graph.NodeNotFoundError
        method), 203
__init__(django_evolution.utils.migrations.MigrationExecutor
        method), 213
__init__(django_evolution.utils.migrations.MigrationList
        method), 209
__init__(django_evolution.utils.migrations.MigrationLoader
        method), 213
__init__(django_evolution.utils.sql.BaseGroupedSQL
        method), 220
__init__(django_evolution.utils.sql.SQLExecutor
        method), 220
__iter__(django_evolution.compat.datastructures.OrderedDict
        method), 150
__iter__(django_evolution.utils.migrations.MigrationList
        method), 212
__iter__(django_evolution.compat.datastructures.OrderedDict
        method), 151
__iter__(django_evolution.utils.migrations.MigrationList
        method), 211
__iter__(django_evolution.compat.datastructures.OrderedDict
        method), 152
__iter__(django_evolution.compat.datastructures.OrderedDict
        method), 151
__iter__(django_evolution.signature.BaseSignature
        method), 114

```

`__new__(django_evolution.compat.picklers.SortedDict static method), 164`  
`__reduce__(django_evolution.compat.datastructures.OrderedDict method), 151`  
`__repr__(django_evolution.compat.datastructures.OrderedDict method), 151`  
`__repr__(django_evolution.db.sql_result.AlterTableSQLResult method), 185`  
`__repr__(django_evolution.db.sql_result.SQLResult method), 185`  
`__repr__(django_evolution.db.state.IndexState method), 192`  
`__repr__(django_evolution.evolve.base.BaseEvolutionTask method), 64`  
`__repr__(django_evolution.mock_models.MockModel method), 138`  
`__repr__(django_evolution.mutations.base.BaseMutation all method), 85`  
`__repr__(django_evolution.placeholders.BasePlaceholder method), 145`  
`__repr__(django_evolution.signature.AppSignature method), 121`  
`__repr__(django_evolution.signature.BaseSignature method), 114`  
`__repr__(django_evolution.signature.ConstraintSignature method), 127`  
`__repr__(django_evolution.signature.FieldSignature method), 132`  
`__repr__(django_evolution.signature.IndexSignature method), 129`  
`__repr__(django_evolution.signature.ModelSignature method), 126`  
`__repr__(django_evolution.signature.ProjectSignature method), 117`  
`__repr__(django_evolution.utils.graph.Node method), 204`  
`__repr__(django_evolution.utils.migrations.MigrationList method), 212`  
`__reversed__(django_evolution.compat.datastructures.OrderedDict method), 150`  
`__set__(django_evolution.compat.models.GenericForeignKey method), 160`  
`__setitem__(django_evolution.compat.datastructures.OrderedDict method), 150`  
`__sizeof__(django_evolution.compat.datastructures.OrderedDict method), 151`  
`__slotnames__(django_evolution.models.VersionManager attribute), 75`  
`__str__(django_evolution.compat.models.GenericForeignKey method), 160`  
`__str__(django_evolution.diff.Diff method), 134`  
`__str__(django_evolution.errors.EvolutionException method), 59`  
`__str__(django_evolution.evolve.base.BaseEvolutionTask method), 65`  
`__str__(django_evolution.evolve.evolve_app_task.EvolveAppTask method), 73`  
`__str__(django_evolution.evolve.purge_app_task.PurgeAppTask method), 74`  
`__str__(django_evolution.models.Evolution method), 77`  
`__str__(django_evolution.models.Version method), 76`  
`__str__(django_evolution.mutations.base.BaseMutation method), 85`  
`__sub__(django_evolution.utils.migrations.MigrationList method), 212`  
`---delete evolution-project-sig command line option, 23`  
`mark-evolution-applied command line option, 26`  
`--app-label mark-evolution-applied command line option, 26`  
`wipe-evolution command line option, 27`  
`--database evolve command line option, 25`  
`--execute evolve command line option, 25`  
`--hint evolve command line option, 25`  
`--id evolution-project-sig command line option, 23`  
`--list evolution-project-sig command line option, 23`  
`--noinput evolution-project-sig command line option, 24`  
`--sql evolve command line option, 25`  
`--write evolve command line option, 25`  
`-w evolve command line option, 25`

```

evolve command line option, 25
<APP_LABEL...>
    evolve command line option, 25
    list-evolutions command line option, 26

A
add() (django_evolution.db.sql_result.AlterTableSQLResult
       method), 185
add() (django_evolution.db.sql_result.SQLResult
       method), 184
add_alter_table() (django_evolution.db.sql_result.AlterTableSQLResult
                  method), 185
add_app() (django_evolution.signature.ProjectSignature
           method), 115
add_app_sig() (django_evolution.signature.ProjectSignature
               method), 116
add_argument() (django_evolution.compat.commands.OptionParser
                method), 149
add_arguments() (django_evolution.compat.commands.BaseCommand
                 method), 150
add_column() (django_evolution.db.common.BaseEvolutionOperation
              method), 170
add_column() (django_evolution.db.sqlite3.EvolutionOperations
              method), 187
add_column() (django_evolution.mutations.add_field.AddField
              method), 79
add_column() (django_evolution.mutators.model_mutator.ModelMutator
              method), 142
add_constraint() (django_evolution.signature.ModelSignature
                  method), 123
add_constraint_sig() (django_evolution.signature.ModelSignature
                      method), 123
add_dependency() (django_evolution.utils.graph.DependencyGraph
                  method), 205
add_evolutions() (django_evolution.utils.graph.EvolutionGraph
                  method), 207
add_field() (django_evolution.signature.ModelSignature
              method), 122
add_field_sig() (django_evolution.signature.ModelSignature
                  method), 123
add_index() (django_evolution.db.state.DatabaseState
              method), 193
add_index() (django_evolution.signature.ModelSignature
              method), 123
add_index_sig() (django_evolution.signature.ModelSignature
                  method), 124
add_m2m_table() (django_evolution.db.common.BaseEvolutionOperation
                  method), 169
add_m2m_table() (django_evolution.mutations.add_field.AddField
                  method), 79
add_migration() (django_evolution.utils.migrations.MigrationList
                 method), 210
add_migration_info() (django_evolution.utils.migrations.MigrationList
                      method), 210
add_migration_plan() (django_evolution.utils.graph.EvolutionGraph
                      method), 207
add_migration_targets() (django_evolution.utils.migrations.MigrationList
                        method), 210
add_model() (django_evolution.signature.AppSignature
              method), 119
add_model_sig() (django_evolution.signature.AppSignature
                  method), 119
add_node() (django_evolution.utils.graph.DependencyGraph
            method), 205
add_post_sql() (django_evolution.db.sql_result.SQLResult
                 method), 185
add_pre_sql() (django_evolution.db.sql_result.SQLResult
                 method), 185
add_recorded_migration()
add_sql() (django_evolution.db.sql_result.SQLResult
            method), 185
add_sql() (django_evolution.mutators.app_mutator.AppMutator
            method), 140
add_sql() (django_evolution.mutators.model_mutator.ModelMutator
            method), 143
add_table() (django_evolution.db.state.DatabaseState
              method), 193
AddField (built-in class), 15
AddField (class in django_evolution.mutations.add_field),
          78
alter_field_type_map (django_evolution.db.postgresql.EvolutionOperations
                      attribute), 182
alter_table_sql_result_cls (django_evolution.db.common.BaseEvolutionOperations
                            attribute), 166
alter_table_sql_result_cls (django_evolution.db.sqlite3.EvolutionOperations
                            attribute), 186
AlterTableSQLResult (class in
                     django_evolution.db.sql_result), 185
APP (django_evolution.consts.EvolutionsSource
      attribute), 57
app (django_evolution.evolve.evolve_app_task.EvolveAppTask
      attribute), 70
app_label (django_evolution.errors.EvolutionExecutionError
            attribute), 70
app_label (django_evolution.evolve.evolve_app_task.EvolveAppTask
            attribute), 70
app_label (django_evolution.evolve.purge_app_task.PurgeAppTask
            attribute), 73

```

<code>app_label</code> ( <i>django_evolution.models.Evolution</i> attribute), 77	<code>attribution)</code> , 77	<code>BaseSignature</code> ( <i>class in django_evolution.signature</i> ), 113
<code>app_sigs</code> ( <i>django_evolution.signature.ProjectSignature</i> property), 115		<code>BaseUpgradeMethodMutation</code> ( <i>class in django_evolution.mutations.base</i> ), 85
<code>applied_evolution</code> ( <i>in django_evolution.signals</i> ), 109	<code>module</code>	<code>build_column_schema()</code> ( <i>django_evolution.db.common.BaseEvolutionOperations method</i> ), 168
<code>applied_migration</code> ( <i>in django_evolution.signals</i> ), 110	<code>module</code>	<code>build_graph()</code> ( <i>django_evolution.utils.migrations.MigrationLoader</i> <i>method</i> ), 213
<code>applied_migrations</code> ( <i>django_evolution.signature.AppSignature</i> property), 119		<code>BUILTIN</code> ( <i>django_evolution.consts.EvolutionsSource</i> attribute), 57
<code>applied_migrations</code> ( <i>django_evolution.utils.migrations</i> . <i>MigrationLoader</i> property), 213		<code>bulk_related_objects()</code> ( <i>django_evolution.compat.models.GenericRelation</i> <i>method</i> ), 161
<code>apply_migrations()</code> ( <i>in django_evolution.utils.migrations</i> ), 216		
<code>apply_unique_together()</code> ( <i>django_evolution.signature.ModelSignature</i> <i>method</i> ), 124		<b>C</b>
<code>applying_evolution</code> ( <i>in django_evolution.signals</i> ), 109	<code>module</code>	<code>can_add_index()</code> ( <i>django_evolution.db.common.BaseEvolutionOperations method</i> ), 167
<code>applying_migration</code> ( <i>in django_evolution.signals</i> ), 110	<code>module</code>	<code>can_simulate()</code> ( <i>django_evolution.evolve.base.BaseEvolutionTask</i> attribute), 62
<code>AppMutator</code> ( <i>class in django_evolution.mutators.app_mutator</i> ), 139		<code>can_simulate()</code> ( <i>django_evolution.evolve.evolver.Evolver</i> <i>method</i> ), 67
<code>AppSignature</code> ( <i>class in django_evolution.signature</i> ), 118		<code>CannotSimulate</code> , 60
<code>atomic()</code> ( <i>in module django_evolution.compat.db</i> ), 152		<code>change_column()</code> ( <i>django_evolution.mutators.model_mutator.ModelMutator</i> <i>method</i> ), 142
<code>auto_created</code> ( <i>django_evolution.compat.models.GenericForeignKey</i> attribute), 159		<code>change_column_attr_db_column()</code>
<code>auto_created</code> ( <i>django_evolution.compat.models.GenericRelation</i> attribute), 160		<code>change_column_attr_db_index()</code> ( <i>django_evolution.db.common.BaseEvolutionOperations method</i> ), 172
		<code>change_column_attr_db_index()</code> ( <i>django_evolution.db.common.BaseEvolutionOperations method</i> ), 173
		<code>change_column_attr_db_index()</code> ( <i>django_evolution.db.sqlite3.EvolutionOperations method</i> ), 190
<b>B</b>		<code>change_column_attr_db_table()</code> ( <i>django_evolution.db.common.BaseEvolutionOperations method</i> ), 173
<code>BaseCommand</code> ( <i>class django_evolution.compat.commands</i> ), 149	<code>in</code>	<code>change_column_attr_decimal_type()</code> ( <i>django_evolution.db.common.BaseEvolutionOperations method</i> ), 172
<code>BaseEvolutionOperations</code> ( <i>class django_evolution.db.common</i> ), 166	<code>in</code>	<code>change_column_attr_decimal_type()</code> ( <i>django_evolution.db.postgresql.EvolutionOperations method</i> ), 183
<code>BaseEvolutionTask</code> ( <i>class django_evolution.evolve.base</i> ), 62	<code>in</code>	<code>change_column_attr_decimal_type()</code> ( <i>django_evolution.db.sqlite3.EvolutionOperations method</i> ), 188
<code>BaseGroupedSQL</code> ( <i>class in django_evolution.utils.sql</i> ), 219		<code>change_column_attr_max_length()</code> ( <i>django_evolution.db.common.BaseEvolutionOperations method</i> ), 172
<code>BaseIterableSerialization</code> ( <i>class django_evolution.serialization</i> ), 100	<code>in</code>	<code>change_column_attr_max_length()</code> ( <i>django_evolution.db.mysql.EvolutionOperations method</i> ), 180
<code>BaseMigrationError</code> , 61		<code>change_column_attr_max_length()</code> ( <i>django_evolution.db.sqlite3.EvolutionOperations method</i> ), 190
<code>BaseModelFieldMutation</code> ( <i>class django_evolution.mutations.base</i> ), 87	<code>in</code>	
<code>BaseModelMutation</code> ( <i>class django_evolution.mutations.base</i> ), 86	<code>in</code>	
<code>BaseMutation</code> ( <i>class django_evolution.mutations.base</i> ), 82	<code>in</code>	
<code>BasePlaceholder</code> ( <i>class django_evolution.placeholders</i> ), 145	<code>in</code>	
<code>BaseRemovedInDjangoEvolutionWarning</code> , 58		
<code>BaseSerialization</code> ( <i>class django_evolution.serialization</i> ), 99	<code>in</code>	

```

    method), 189
change_column_attr_null()                                89
    (django_evolution.db.common.BaseEvolutionOperations
     method), 172
change_column_attr_null()                                check() (django_evolution.compat.models.GenericForeignKey
    (django_evolution.db.sqlite3.EvolutionOperations
     child_separators (django_evolution.serialization.QSerialization
     method), 188
     attribute), 107
change_column_attr_unique()                            ClassSerialization      (class      in
    (django_evolution.db.common.BaseEvolutionOperations
     django_evolution.serialization), 102
     method), 174
change_column_attrs()                                 clear() (django_evolution.compat.datastructures.OrderedDict
    (django_evolution.db.common.BaseEvolutionOperations
     method), 172
     method), 151
change_column_attrs_db_index_unique()                  clear_app_cache()      (in      module
    (django_evolution.db.common.BaseEvolutionOperations
     django_evolution.utils.migrations), 214
     method), 173
     module), 148
change_column_type()                                  clear_global_custom_migrations() (in module
    (django_evolution.db.common.BaseEvolutionOperations
     django_evolution.utils.models), 218
     method), 174
change_column_type()                                  clear_indexes() (django_evolution.db.state.DatabaseState
    (django_evolution.db.common.BaseEvolutionOperations
     method), 194
     method), 192
change_column_type()                                  clear_model_sigs() (django_evolution.signature.AppSignature
    (django_evolution.db.common.BaseEvolutionOperations
     method), 119
     method), 189
change_column_type()                                  clone()      (django_evolution.db.state.DatabaseState
    (django_evolution.mutators.model_mutator.ModelMutator
     method), 120
     method), 192
change_column_type_setsAttrs()                        clone()      (django_evolution.signature.BaseSignature
    (django_evolution.db.common.BaseEvolutionOperations
     method), 114
     attribute), 166
     method), 127
change_column_type_setsAttrs()                        clone()      (django_evolution.signature.ConstraintSignature
    (django_evolution.db.common.BaseEvolutionOperations
     method), 131
     attribute), 182
change_meta() (django_evolution.mutators.model_mutator.ModelMutator
     method), 143
     method), 128
change_meta_constraints()                            clone()      (django_evolution.signature.ModelSignature
    (django_evolution.db.common.BaseEvolutionOperations
     method), 125
     method), 175
     method), 125
change_meta_db_table_comment()                      clone()      (django_evolution.signature.ProjectSignature
    (django_evolution.db.common.BaseEvolutionOperations
     method), 211
     method), 117
change_meta_index_together()                        CombinedExpressionSerialization (class      in
    (django_evolution.db.common.BaseEvolutionOperations
     django_evolution.serialization), 106
     method), 175
     attribute), 159
change_meta_indexes()                             connection (django_evolution.evolve.evolver.Evolver
    (django_evolution.db.common.BaseEvolutionOperations
     attribute), 65
     method), 176
change_meta_unique_together()                      ConstraintSignature      (class      in
    (django_evolution.db.common.BaseEvolutionOperations
     django_evolution.signature), 126
     method), 174
     method), 126
changeField (built-in class), 16                    contribute_to_class() (django_evolution.compat.models.GenericForeignKey
changeField          (class      in      method), 160
     django_evolution.mutations.change_field),
     87
     method), 160
changeMeta (built-in class), 18                     contribute_to_class() (django_evolution.compat.models.GenericRelation
changeMeta (class in django_evolution.mutations.change_m
     method), 161
     method), 161
     contribute_to_class())

```

(*django\_evolution.models.SignatureField method*), 75

`copy()` (*django\_evolution.compat.datastructures.OrderedDict method*), 151

`create_constraint_name()` (in *django\_evolution.compat.db*), 153

`create_field()` (in *django\_evolution.mock\_models*), 135

`create_index()` (*django\_evolution.db.common.BaseEvolutionOperations method*), 170

`create_index_name()` (in *django\_evolution.compat.db*), 153

`create_index_together_name()` (in *django\_evolution.compat.db*), 153

`create_model()` (*django\_evolution.mutators.model\_mutator.ModelMutator method*), 142

`create_parser()` (*django\_evolution.compat.commands.BashParser method*), 149

`create_pre_migrate_state()` (in *django\_evolution.utils.migrations*), 215

`create_unique_index()` (in *django\_evolution.db.common.BaseEvolutionOperations method*), 170

`created_models` (in module *django\_evolution.signals*), 110

`creating_models` (in *django\_evolution.signals*), 110

`current_version()` (*django\_evolution.models.VersionManager*)

`CUSTOM_EVOLUTIONS` (*django\_evolution.conf.DjangoEvolution attribute*), 56

**D**

`database_name` (*django\_evolution.evolve.evolver.Evolver attribute*), 65

`database_state` (*django\_evolution.evolve.evolver.Evolver attribute*), 66

`DatabaseState` (class in *django\_evolution.db.state*), 192

`DatabaseStateError`, 60

`db_get_installable_models_for_app()` (in module *django\_evolution.compat.db*), 154

`db_routerAllows_migrate()` (in *django\_evolution.compat.db*), 154

`db_routerAllows_schema_upgrade()` (in *django\_evolution.compat.db*), 154

`db_routerAllows_syncdb()` (in *django\_evolution.compat.db*), 154

`DeconstructedSerialization` (class in *django\_evolution.serialization*), 105

`default_tablespace` (*django\_evolution.db.common.BaseEvolutionOperations attribute*), 166

`default_tablespace` (*django\_evolution.db.postgresql.EvolutionOperations attribute*), 182

`delete_column()` (*django\_evolution.db.common.BaseEvolutionOperations method*), 169

`delete_column()` (*django\_evolution.db.mysql.EvolutionOperations method*), 180

`delete_column()` (*django\_evolution.db.sqlite3.EvolutionOperations method*), 187

`delete_model()` (*django\_evolution.mutators.model\_mutator.ModelMutator method*), 142

`delete_table()` (*django\_evolution.db.common.BaseEvolutionOperations method*), 169

`DeleteApplication` (class in *django\_evolution.mutations.delete\_application*), 16

`DeleteField` (built-in class), 16

`DeleteField` (class in *django\_evolution.mutations.delete\_field*), 91

`DeleteModel` (built-in class), 19

`DeleteModel` (class in *django\_evolution.mutations.delete\_model*), 92

`dependencies` (*django\_evolution.utils.graph.Node attribute*), 204

`DependencyGraph` (class in *django\_evolution.utils.graph*), 205

`deserializer` (*django\_evolution.models.SignatureField attribute*), 75

`deserialize()` (*django\_evolution.signature.AppSignature class method*), 118

`deserialize()` (*django\_evolution.signature.BaseSignature class method*), 113

`deserialize()` (*django\_evolution.signature.ConstraintSignature class method*), 126

`deserialize()` (*django\_evolution.signature.FieldSignature class method*), 130

`deserialize()` (*django\_evolution.signature.IndexSignature class method*), 128

`deserialize()` (*django\_evolution.signature.ModelSignature class method*), 121

`deserialize()` (*django\_evolution.signature.ProjectSignature class method*), 115

`deserialize_from_deconstructed()` (*django\_evolution.serialization.BaseSerialization class method*), 100

`deserialize_from_deconstructed()` (*django\_evolution.serialization.QSerialization class method*), 107

`deserialize_from_signature()`

`deserialize_from_signature()` (*django\_evolution.serialization.BaseIterableSerialization class method*), 101

`deserialize_from_signature()` (*django\_evolution.serialization.BaseSerialization class method*), 107

```
    class method), 100
deserialize_from_signature()
    (django_evolution.serialization.DeconstructedSerialization
        class method), 106
deserialize_from_signature()
    (django_evolution.serialization.DictSerialization
        class method), 103
deserialize_from_signature()
    (django_evolution.serialization.EnumSerialization
        class method), 103
deserialize_from_signature()
    (django_evolution.serialization.PrimitiveSerialization
        class method), 101
deserialize_from_signature()
    (in     module django_evolution.serialization), 108
detailed_error(django_evolution.errors.EvolutionExecutionError,
    attribute), 60
DictSerialization      (class django_evolution.serialization), 102
Diff (class in django_evolution.diff), 133
diff()    (django_evolution.signature.AppSignature
    method), 120
diff()    (django_evolution.signature.BaseSignature
    method), 113
diff()    (django_evolution.signature.FieldSignature
    method), 131
diff()    (django_evolution.signature.ModelSignature
    method), 124
diff()    (django_evolution.signature.ProjectSignature
    method), 116
diff_evolutions()(django_evolution.evolve.evolver.Evolver
    method), 67
digest()(in module django_evolution.compat.db), 155
django_evolution
    module, 55
django_evolution.compat.apps
    module, 147
django_evolution.compat.commands
    module, 149
django_evolution.compat.datastructures
    module, 150
django_evolution.compat.db
    module, 152
django_evolution.compat.models
    module, 159
django_evolution.compat.picklers
    module, 164
django_evolution.compat.py23
    module, 165
django_evolution.conf
    module, 56
django_evolution.consts
    module, 57
django_evolution.db.common
    module, 166
django_evolution.db.mysql
    module, 179
django_evolution.db.postgresql
    module, 182
django_evolution.db.sqlite3
    module, 186
django_evolution.db.state
    module, 191
django_evolution.deprecation
    module, 58
django_evolution.diff
    module, 133
django_evolution.errors
    module, 59
django_evolution.evolve
    module, 62
django_evolution.evolve.base
    module, 62
django_evolution.evolve.evolve_app_task
    module, 70
django_evolution.evolve.evolver
    module, 65
django_evolution.evolve.purge_app_task
    module, 73
django_evolution.mock_models
    module, 135
django_evolution.models
    module, 74
django_evolution.mutations
    module, 78
django_evolution.mutations.add_field
    module, 78
django_evolution.mutations.base
    module, 80
django_evolution.mutations.change_field
    module, 87
django_evolution.mutations.change_meta
    module, 88
django_evolution.mutations.delete_application
    module, 90
django_evolution.mutations.delete_field
    module, 91
django_evolution.mutations.delete_model
    module, 92
django_evolution.mutations.move_to_django_migrations
    module, 93
django_evolution.mutations.rename_app_label
    module, 94
django_evolution.mutations.rename_field
    module, 95
django_evolution.mutations.rename_model
```

```

    module, 96
django_evolution.mutations.sql_mutation
    module, 97
django_evolution.mutators
    module, 139
django_evolution.mutators.app_mutator
    module, 139
django_evolution.mutators.model_mutator
    module, 141
django_evolution.mutators.sql_mutator
    module, 144
django_evolution.placeholders
    module, 144
django_evolution.serialization
    module, 99
django_evolution.signals
    module, 109
django_evolution.signature
    module, 110
django_evolution.support
    module, 146
django_evolution.utils.apps
    module, 195
django_evolution.utils.datastructures
    module, 196
django_evolution.utils.evolutions
    module, 197
django_evolution.utils.graph
    module, 203
django_evolution.utils.migrations
    module, 208
django_evolution.utils.models
    module, 217
django_evolution.utils.sql
    module, 219
DjangoCompatUnpickler      (class
    django_evolution.compat.picklers), 164
DjangoEvolutionSettings     (class
    django_evolution.conf), 56
DjangoEvolutionSupportError, 61
drop_index() (django_evolution.db.common.BaseEvolutionOperations
    method), 170
drop_index_by_name()
    (django_evolution.db.common.BaseEvolutionOperations
        method), 170

E
editable(django_evolution.compat.models.GenericForeignKey
    attribute), 159
emit_post_migrate_or_sync() (in module
    django_evolution.utils.migrations), 217
emit_pre_migrate_or_sync() (in module
    django_evolution.utils.migrations), 216

ENABLED (django_evolution.conf.DjangoEvolutionSettings
    attribute), 56
ensure_transaction()
    (django_evolution.utils.sql.SQLExecutor
        method), 221
EnumSerialization           (class
    in django_evolution.serialization), 103
error_vars(django_evolution.mutations.base.BaseModelFieldMutation
    attribute), 87
error_vars(django_evolution.mutations.base.BaseModelMutation
    attribute), 86
error_vars(django_evolution.mutations.base.BaseMutation
    attribute), 82
error_vars(django_evolution.mutations.change_meta.ChangeMeta
    attribute), 89
Evolution (class in django_evolution.models), 77
evolution label, 29
evolution() (django_evolution.diff.Diff method), 135
EVOLUTION_LABEL
    mark-evolution-applied command line
        option, 26
    wipe-evolution command line option, 27
evolution_required(django_evolution.evolve.base.BaseEvolutionTask
    attribute), 62
evolution-project-sig command line option
    --delete, 23
    --id, 23
    --list, 23
    --noinput, 24
    --show, 23
EvolutionBaselineMissingError, 61
EvolutionException, 59
EvolutionExecutionError, 59
EvolutionGraph            (class
    in django_evolution.utils.graph), 206
in EvolutionNotImplementedError, 60
EvolutionOperations        (class
    in django_evolution.db.mysql), 179
EvolutionOperations        (class
    in django_evolution.db.postgresql), 182
EvolutionOperations        (class
    in django_evolution.db.sqlite3), 186
EVOLUTIONS (django_evolution.consts.UpgradeMethod
    attribute), 57
evolutions (django_evolution.models.Version
    attribute), 77
EvolutionsSource (class in django_evolution.consts),
    57
EvolutionTaskAlreadyQueuedError, 61
evolve command line option
    --database, 25
    --execute, 25
    --hint, 25
    --noinput, 25

```

```
--purge, 25
--sql, 25
--write, 25
-w, 25
-x, 25
<APP_LABEL...>, 25
evolve() (django_evolution.evolve.evolver.Evolver
           method), 69
EvolveAppTask (class in
               django_evolution.evolve.evolve_app_task),
              70
evolved (django_evolution.evolve.evolver.Evolver
          attribute), 66
evolved (in module django_evolution.signals), 109
Evolver (class in django_evolution.evolve.evolver), 65
evolver (django_evolution.evolve.base.BaseEvolutionTask
          attribute), 62
evolver() (django_evolution.mutations.base.BaseModelMutation
            method), 86
evolving (in module django_evolution.signals), 109
evolving_failed (in module
                  django_evolution.signals), 109
execute() (django_evolution.evolve.base.BaseEvolutionTask
            method), 64
execute() (django_evolution.evolve.evolve_app_task.EvolveAppTask
            class method), 128
execute() (django_evolution.evolve.purge_app_task.PurgeAppTask
            method), 73
execute() (django_evolution.evolve.base.BaseEvolutionTask
            class method), 209
execute() (django_evolution.evolve.base.BaseEvolutionTask
            class method), 63
execute_tasks() (django_evolution.evolve.evolve_app_task.EvolveAppTask,
                  151
                  class method), 70
extra_applied_migrations
    (django_evolution.utils.migrations.MigrationLoader
     attribute), 213
F
fail() (django_evolution.mutations.base.Simulation
        method), 81
field_sigs (django_evolution.signature.ModelSignature
            property), 122
FieldDoesNotExist, 159
fields (django_evolution.mock_models.MockMeta
        property), 136
FieldSignature (class in django_evolution.signature),
               129
filter_dup_list_items() (in module
                         django_evolution.utils.datastructures), 196
filter_migration_targets() (in module
                           django_evolution.utils.migrations), 215
finalize() (django_evolution.utils.graph.DependencyGraph
            method), 205
finalize_migrations() (in module
                       django_evolution.utils.migrations), 216
find_class() (django_evolution.compat.picklers.DjangoCompatUnpickler
              method), 164
find_index() (django_evolution.db.state.DatabaseState
              method), 194
finish_op() (django_evolution.mutators.model_mutator.ModelMutator
              method), 143
finish_transaction()
    (django_evolution.utils.sql.SQLExecutor
     method), 221
from_app() (django_evolution.signature.AppSignature
            class method), 118
from_app_sig() (django_evolution.utils.migrations.MigrationList
                class method), 209
from_constraint() (django_evolution.signature.ConstraintSignature
                   class method), 126
from_database() (django_evolution.signature.ProjectSignature
                 class method), 115
from_database() (django_evolution.utils.migrations.MigrationList
                 class method), 209
from_evolver() (django_evolution.mutators.app_mutator.AppMutator
                 class method), 140
from_field() (django_evolution.signature.FieldSignature
               class method), 130
from_index() (django_evolution.signature.IndexSignature
               class method), 128
from_model() (django_evolution.signature.ModelSignature
               class method), 121
from_names() (django_evolution.utils.migrations.MigrationList
               class method), 209
fromkeys() (django_evolution.compat.datastructures.OrderedDict
            method), 93
G
generate_dependencies()
    (django_evolution.mutations.base.BaseMutation
     method), 82
generate_dependencies()
    (django_evolution.mutations.base.BaseUpgradeMethodMutation
     method), 85
generate_dependencies()
    (django_evolution.mutations.move_to_django_migrations.MoveTo
     method), 93
generate_hint() (django_evolution.mutations.base.BaseMutation
                 method), 82
generate_mutations_info()
    (django_evolution.evolve.evolve_app_task.EvolveAppTask
     method), 71
generate_table_op_sql()
    (django_evolution.db.common.BaseEvolutionOperations
     method), 168
generate_table_ops_sql()
    (django_evolution.db.common.BaseEvolutionOperations
     method), 168
```

```

GenericForeignKey      (class      in      method), 177
                     django_evolution.compat.models), 159
GenericRelation       (class      in      get_constraints_for_table()
                     django_evolution.compat.models), 160
                     (django_evolution.db.common.BaseEvolutionOperations
                      method), 177
get_app()             (in module django_evolution.compat.apps), 148
get_app_config_for_app() (in module django_evolution.utils.apps), 195
get_app_label()        (in module django_evolution.utils.apps), 195
get_app_labels()       (django_evolution.utils.migrations.Migration)
                     get_prep_value()
                     (method), 211
get_app_mutations()   (in module django_evolution.utils.evolutions), 201
get_app_name()         (in module django_evolution.utils.apps), 195
get_app_pending_mutations() (in module django_evolution.utils.evolutions), 201
get_app_sig()          (django_evolution.mutations.base.Simulation
                     method), 80
get_app_sig()          (django_evolution.signature.ProjectSignature
                     method), 116
get_app_upgrade_info() (in module django_evolution.utils.evolutions), 202
get_applied_evolutions() (in module django_evolution.utils.evolutions), 201
get_apps()              (in module django_evolution.compat.apps), 148
get_attr_default()     (django_evolution.signature.FieldSignature
                     method), 131
get_attr_value()        (django_evolution.signature.FieldSignature
                     method), 130
get_cache_name()        (django_evolution.compat.models.GenericForeignKey
                     method), 160
get_change_column_type_sql() (django_evolution.db.common.BaseEvolutionOperations
                           method), 167
get_change_column_type_sql() (django_evolution.db.mysql.EvolutionOperations
                           method), 180
get_change_column_type_sql() (django_evolution.db.postgresql.EvolutionOperations
                           method), 180
get_change_column_type_sql() (django_evolution.db.sqlite3.EvolutionOperations
                           method), 182
get_change_unique_sql()  (django_evolution.db.common.BaseEvolutionOperations
                     method), 174
get_change_unique_sql()  (django_evolution.db.mysql.EvolutionOperations
                     method), 181
get_change_unique_sql()  (django_evolution.db.sqlite3.EvolutionOperations
                     method), 189
get_change_unique_sql()  (django_evolution.db.common.BaseEvolutionOperations
                     method), 189
get_change_unique_sql()  (django_evolution.db.mysql.EvolutionOperations
                     method), 181
get_change_unique_sql()  (django_evolution.db.sqlite3.EvolutionOperations
                     method), 190
get_change_unique_sql()  (django_evolution.evolve.base.BaseEvolutionTask
                     method), 64
get_change_unique_sql()  (django_evolution.evolve.evolve_app_task.EvolveAppTask
                     method), 72
get_column_names_for_fields() (django_evolution.db.common.BaseEvolutionOperations
                           method), 200
get_content_type()      (django_evolution.compat.models.GenericForeignKey
                     method), 160
get_content_type()      (django_evolution.compat.models.GenericRelation
                     method), 161
get_database_for_model_name() (in module django_evolution.utils.models), 217
get_default_index_name() (django_evolution.db.common.BaseEvolutionOperations
                     method), 171
get_default_index_name() (django_evolution.db.mysql.EvolutionOperations
                     method), 181
get_default_index_name() (django_evolution.db.postgresql.EvolutionOperations
                     method), 183
get_default_index_together_name() (django_evolution.db.common.BaseEvolutionOperations
                     method), 171
get_deferrable_sql()    (django_evolution.db.common.BaseEvolutionOperations
                     method), 167
get_deferrable_sql()    (django_evolution.db.sqlite3.EvolutionOperations
                     method), 186
get_drop_index_sql()    (django_evolution.db.common.BaseEvolutionOperations
                     method), 170
get_drop_index_sql()    (django_evolution.db.mysql.EvolutionOperations
                     method), 180
get_drop_index_sql()    (django_evolution.db.postgresql.EvolutionOperations
                     method), 183
get_drop_index_sql()    (django_evolution.db.sqlite3.EvolutionOperations
                     method), 190
get_drop_unique_constraint_sql() (django_evolution.db.common.BaseEvolutionOperations
                     method), 174
get_drop_unique_constraint_sql() (django_evolution.db.mysql.EvolutionOperations
                     method), 174
get_drop_unique_constraint_sql() (django_evolution.db.postgresql.EvolutionOperations
                     method), 183
get_drop_unique_constraint_sql() (django_evolution.db.sqlite3.EvolutionOperations
                     method), 190
get_evolution_app_dependencies() (in module django_evolution.utils.evolutions), 200
get_evolution_content()   (django_evolution.evolve.base.BaseEvolutionTask
                     method), 64
get_evolution_content()   (django_evolution.evolve.evolve_app_task.EvolveAppTask
                     method), 72

```

```

get_evolution_dependencies()      (in module django_evolution.utils.evolutions), 199
get_evolution_module()          (in module django_evolution.utils.evolutions), 198
get_evolution_required()
    (django_evolution.evolve.evolver.Evolver method), 67
get_evolution_sequence()        (in module django_evolution.utils.evolutions), 199
get_evolutions_module()         (in module django_evolution.utils.evolutions), 198
get_evolutions_module_name()    (in module django_evolution.utils.evolutions), 198
get_evolutions_path()           (in module django_evolution.utils.evolutions), 198
get_evolutions_source()         (in module django_evolution.utils.evolutions), 197
get_evolver() (django_evolution.mutations.base.Simulation method), 80
get_extra_restriction()
    (django_evolution.compat.models.GenericRelation method), 161
get_field() (django_evolution.mock_models.MockMeta method), 137
get_field_by_name()
    (django_evolution.mock_models.MockMeta method), 137
get_field_is_hidden()          (in module django_evolution.compat.models), 161
get_field_is_many_to_many()    (in module django_evolution.compat.models), 161
get_field_is_relation()        (in module django_evolution.compat.models), 161
get_field_sig() (django_evolution.mutations.base.Simulation method), 81
get_field_sig() (django_evolution.signature.ModelSignature method), 123
get_field_typeAllows_default()
    (django_evolution.db.common.BaseEvolutionOperations method), 167
get_field_typeAllows_default()
    (django_evolution.db.mysql.EvolutionOperations method), 179
get_fields_for_names()
    (django_evolution.db.common.BaseEvolutionOperations method), 177
get_filter_kwargs_for_object()
    (django_evolution.compat.models.GenericForeignKey method), 160
get_forward_related_filter()
    (django_evolution.compat.models.GenericForeignKey method), 160
get_hint_params() (django_evolution.mutations.add_field.AddField method), 79
get_hint_params() (django_evolution.mutations.base.BaseMutation method), 82
get_hint_params() (django_evolution.mutations.change_field.ChangeField method), 88
get_hint_params() (django_evolution.mutations.change_meta.ChangeMeta method), 89
get_hint_params() (django_evolution.mutations.delete_field.DeleteField method), 91
get_hint_params() (django_evolution.mutations.delete_model.DeleteModel method), 92
get_hint_params() (django_evolution.mutations.rename_app_label.RenameAppLabel method), 94
get_hint_params() (django_evolution.mutations.rename_field.RenameField method), 95
get_hint_params() (django_evolution.mutations.rename_model.RenameModel method), 96
get_hint_params() (django_evolution.mutations.sql_mutation.SQLMutation method), 98
get_index() (django_evolution.db.state.DatabaseState method), 194
get_indexes_for_table()
    (django_evolution.db.common.BaseEvolutionOperations method), 177
get_indexes_for_table()
    (django_evolution.db.mysql.EvolutionOperations method), 181
get_indexes_for_table()
    (django_evolution.db.postgresql.EvolutionOperations method), 183
get_indexes_for_table()
    (django_evolution.db.sqlite3.EvolutionOperations method), 191
get_internal_type()
    (django_evolution.compat.models.GenericRelation method), 161
get_leaf_nodes() (django_evolution.utils.graph.DependencyGraph method), 206
get_legacy_app_label()          (in module django_evolution.utils.apps), 196
get_model() (in module django_evolution.compat.models), 162
get_model_name() (in module django_evolution.compat.models), 162
get_model_rel_tree() (in module django_evolution.utils.models), 218
get_model_sig() (django_evolution.mutations.base.Simulation method), 81
get_model_sig() (django_evolution.signature.AppSignature method), 119
get_models() (in module django_evolution.compat.models), 162
get_new_constraint_name()
    (django_evolution.db.common.BaseEvolutionOperations method), 171

```

```

get_new_index_name()           has_migrations_module()      (in      module
                            (django_evolution.db.common.BaseEvolutionOperations django_evolution.utils.migrations), 214
                            method), 170
get_next_by_when() (django_evolution.models.Version      has_model() (django_evolution.db.state.DatabaseState
                     method), 77                           method), 193
get_node() (django_evolution.utils.graph.DependencyGraph      has_table() (django_evolution.db.state.DatabaseState
                     method), 205                           method), 193
get_ordered() (django_evolution.utils.graph.DependencyGraph      has_unique_together_changed()
                     method), 206                           (django_evolution.signature.ModelSignature
get_package_version() (in      module      attribute), 124
                            django_evolution), 56
get_path_info() (django_evolution.compat.models.GenericForeignKey      hasRelation(django_evolution.evolve.Evolver
                     attribute), 159
                     method), 160                           attribute), 66
get_prefetch_queryset()
    (django_evolution.compat.models.GenericForeignKey
     method), 160
get_prep_value() (django_evolution.models.SignatureField      id (django_evolution.evolve.base.BaseEvolutionTask at-
                     method), 75                           tribute), 62
get_previous_by_when()
    (django_evolution.models.Version method), 77
get_rel_target_field() (in      module      id (django_evolution.models.Evolution attribute), 77
                            django_evolution.compat.models), 162
get_remote_field() (in      module      id (django_evolution.models.Version attribute), 77
                            django_evolution.compat.models), 163
get_remote_field_model() (in      module      ignored_m2mAttrs(django_evolution.db.common.BaseEvolutionOperat-
                            django_evolution.compat.models), 163
get_remote_field_related_model() (in      module      attribute), 166
                            django_evolution.compat.models), 163
get_rename_table_sql()
    (django_evolution.db.common.BaseEvolutionOperations      import_management_modules() (in      module
                     method), 169                           django_evolution.utils.apps), 196
get_rename_table_sql()
    (django_evolution.db.mysql.EvolutionOperations      index_together(django_evolution.signature.ModelSignature
                     method), 181                           property), 122
get_reverse_path_info()
    (django_evolution.compat.models.GenericRelation      IndexSignature (class in django_evolution.signature),
                     method), 160                           128
get_unapplied_evolutions() (in      module      IndexState (class in django_evolution.db.state), 191
                            django_evolution.utils.evolutions), 200
get_update_table_constraints_sql()
    (django_evolution.db.common.BaseEvolutionOperations      initial_diff (django_evolution.evolve.Evolver
                     method), 176                           attribute), 66
get_update_table_constraints_sql()
    (django_evolution.db.sqlite3.EvolutionOperations      insert_index (django_evolution.utils.graph.Node at-
                     method), 189                           tribute), 204
get_version_string() (in module django_evolution,
                     56                           interactive (django_evolution.evolve.Evolver
                                         attribute), 66
                                         InvalidSignatureVersion, 61
                                         is_attr_value_default()
                                         (django_evolution.signature.FieldSignature
                                         method), 131
                                         is_column_referenced()
                                         (django_evolution.db.sqlite3.EvolutionOperations
                                         method), 191
                                         is_empty() (django_evolution.diff.Diff method), 134
                                         is_empty() (django_evolution.signature.AppSignature
                                         method), 119
                                         is_hinted() (django_evolution.models.Version
                                         method), 76
                                         is_migration_initial() (in      module
                                         django_evolution.utils.migrations), 215
                                         is_mutable() (django_evolution.mutations.base.BaseModelMutation
                                         method), 86
                                         is_mutable() (django_evolution.mutations.base.BaseMutation
                                         method), 83
                                         is_mutable() (django_evolution.mutations.base.BaseUpgradeMethodMutation
                                         method), 85
H
has_evolutions_module() (in      module
                            django_evolution.utils.evolutions), 197
has_migration_info()
    (django_evolution.utils.Migrations.MigrationList
     method), 210

```

is Mutable() (*django\_evolution.mutations.delete\_application*)  
    method), 90  
is Mutable() (*django\_evolution.mutations.rename\_app\_label*)  
    method), 94  
is Mutable() (*django\_evolution.mutations.sql\_mutation*)  
    method), 98  
is mutation Mutable()  
    (django\_evolution.evolve.base.BaseEvolutionTask  
        method), 63  
is relation (*django\_evolution.compat.models.GenericForeignKey*)  
    attribute), 159  
is release() (in module *django\_evolution*), 56  
item type (*django\_evolution.serialization.BaseIterableSerialization*)  
    attribute), 100  
item type (*django\_evolution.serialization.ListSerialization*)  
    attribute), 104  
item type (*django\_evolution.serialization.SetSerialization*)  
    attribute), 104  
item type (*django\_evolution.serialization.TupleSerialization*)  
    attribute), 104  
items() (*django\_evolution.compat.datastructures.OrderedDict*)  
    method), 151  
iter batches() (*django\_evolution.utils.graph.EvolutionGraph*)  
    method), 208  
iter evolution content()  
    (django\_evolution.evolve.evolver.Evolver  
        method), 67  
iter indexes() (*django\_evolution.db.state.DatabaseState*)  
    method), 194  
iter model fields() (in module *django\_evolution.utils.models*), 218  
iter non m2m reverse relations() (in module *django\_evolution.utils.models*), 219

**M**

Mark evolutions applied()  
    (*django\_evolution.utils.graph.EvolutionGraph*  
        method), 207  
mark migrations applied()  
    (*django\_evolution.utils.graph.EvolutionGraph*  
        method), 208  
mark-evolution-applied command line option  
    --all, 26  
    --app-label, 26  
    --noinput, 26  
    EVOLUTION\_LABEL, 26  
merge dicts() (in module *django\_evolution.utils.datastructures*), 196  
mergeable ops (*django\_evolution.db.common.BaseEvolutionOperations*)  
    attribute), 166  
MigrationConflictsError, 61  
MigrationExecutor (class in *django\_evolution.utils.migrations*), 213  
MigrationHistoryError, 61  
MigrationList (class in *django\_evolution.utils.migrations*), 209  
MigrationLoader (class in *django\_evolution.utils.migrations*), 212  
migrations, 29  
MIGRATIONS (*django\_evolution.consts.UpgradeMethod*)  
    attribute), 57  
MissingSignatureError, 60  
MockMeta (class in *django\_evolution.mock\_models*), 136  
MockModel (class in *django\_evolution.mock\_models*), 137  
MockRelated (class in *django\_evolution.mock\_models*), 139  
model sig (*django\_evolution.mutators.model\_mutator.ModelMutator*)  
    property), 142  
model sigs (*django\_evolution.signature.AppSignature*)  
    property), 119

**K**

key (*django\_evolution.utils.graph.Node* attribute), 204  
keys() (*django\_evolution.compat.datastructures.OrderedDict*)  
    method), 151

**L**

label (*django\_evolution.mock\_models.MockMeta* property), 136  
label (*django\_evolution.models.Evolution* attribute), 77  
last sql statement (*django\_evolution.errors.EvolutionExecutionError*)  
    attribute), 60  
LATEST\_SIGNATURE\_VERSION (in module *django\_evolution.signature*), 113  
legacy app label, 29  
legacy app labels, 29  
list evolutions command line option  
    <APP\_LABEL...>, 26  
ListSerialization (class in *django\_evolution.serialization*), 103

`ModelMutator` (class in `django_evolution.mutators.model_mutator`), 141  
`ModelSignature` (class in `django_evolution.signature`), 121  
 modern app label, 29  
 modern app labels, 29  
 module  
   `django_evolution`, 55  
   `django_evolution.compat.apps`, 147  
   `django_evolution.compat.commands`, 149  
   `django_evolution.compat.datastructures`, 150  
   `django_evolution.compat.db`, 152  
   `django_evolution.compat.models`, 159  
   `django_evolution.compat.picklers`, 164  
   `django_evolution.compat.py23`, 165  
   `django_evolution.conf`, 56  
   `django_evolution.consts`, 57  
   `django_evolution.db.common`, 166  
   `django_evolution.db.mysql`, 179  
   `django_evolution.db.postgresql`, 182  
   `django_evolution.db.sql_result`, 184  
   `django_evolution.db.sqlite3`, 186  
   `django_evolution.db.state`, 191  
   `django_evolution.deprecation`, 58  
   `django_evolution.diff`, 133  
   `django_evolution.errors`, 59  
   `django_evolution.evolve`, 62  
   `django_evolution.evolve.base`, 62  
   `django_evolution.evolve.evolve_app_task`, 70  
   `django_evolution.evolve.evolver`, 65  
   `django_evolution.evolve.purge_app_task`, 73  
   `django_evolution.mock_models`, 135  
   `django_evolution.models`, 74  
   `django_evolution.mutations`, 78  
   `django_evolution.mutations.add_field`, 78  
   `django_evolution.mutations.base`, 80  
   `django_evolution.mutations.change_field`, 87  
   `django_evolution.mutations.change_meta`, 88  
   `django_evolution.mutations.delete_application`, 90  
   `django_evolution.mutations.delete_field`, 91  
   `django_evolution.mutations.delete_model`, 92  
   `django_evolution.mutations.move_to_django_migrations`, 93  
   `django_evolution.mutations.rename_app_label`, 94

`djangoevolution.mutations.rename_field`, 95  
`djangoevolution.mutations.rename_model`, 96  
`djangoevolution.mutations.sql_mutation`, 97  
`djangoevolution.mutators`, 139  
`djangoevolution.mutators.app_mutator`, 139  
`djangoevolution.mutators.model_mutator`, 141  
`djangoevolution.mutators.sql_mutator`, 144  
`djangoevolution.placeholders`, 144  
`djangoevolution.serialization`, 99  
`djangoevolution.signals`, 109  
`djangoevolution.signature`, 110  
`djangoevolution.support`, 146  
`djangoevolution.utils.apps`, 195  
`djangoevolution.utils.datastructures`, 196  
`djangoevolution.utils.evolutions`, 197  
`djangoevolution.utils.graph`, 203  
`djangoevolution.utils.migrations`, 208  
`djangoevolution.utils.models`, 217  
`djangoevolution.utils.sql`, 219  
`move_to_end()` (`djangoevolution.compat.datastructures.OrderedDict` method), 151  
`MoveToDjangoMigrations` (built-in class), 20  
`MoveToDjangoMigrations` (class in `djangoevolution.mutations.move_to_django_migrations`), 93  
`mti_inherited` (`djangoevolution.compat.models.GenericRelation` attribute), 160  
`mutate()` (`djangoevolution.mutations.add_field.AddField` method), 79  
`mutate()` (`djangoevolution.mutations.base.BaseModelMutation` method), 86  
`mutate()` (`djangoevolution.mutations.base.BaseMutation` method), 83  
`mutate()` (`djangoevolution.mutations.base.BaseUpgradeMethodMutation` method), 86  
`mutate()` (`djangoevolution.mutations.change_field.ChangeField` method), 88  
`mutate()` (`djangoevolution.mutations.change_meta.ChangeMeta` method), 89  
`mutate()` (`djangoevolution.mutations.delete_application.DeleteApplication` method), 90  
`mutate()` (`djangoevolution.mutations.delete_field.DeleteField` method), 91  
`mutate()` (`djangoevolution.mutations.delete_model.DeleteModel` method), 92  
`mutate()` (`djangoevolution.mutations.rename_app_label.RenameAppLabel` method), 95

`mutate()` (`django_evolution.mutations.rename_field.RenameFields` (`django_evolution.models.Version` attribute), 76  
    method), 96  
`mutate()` (`django_evolution.mutations.rename_model.RenameModel` attribute), 159  
    method), 97  
`mutate()` (`django_evolution.mutations.sql_mutation.SQLMutation` attribute), 160  
    method), 98  
  
**N**  
`name` (`django_evolution.db.common.BaseEvolutionOperations` attribute), 166  
`name` (`django_evolution.db.mysql.EvolutionOperations` attribute), 179  
`name` (`django_evolution.db.postgresql.EvolutionOperations` attribute), 182  
`name` (`django_evolution.db.sqlite3.EvolutionOperations` attribute), 186  
`new_evolutions` (`django_evolution.evolve.base.BaseEvolutionOperations` attribute), 62  
`new_transaction()` (`django_evolution.utils.sql.SQLExecute` method), 221  
`NewTransactionSQL` (class in `django_evolution.utils.sql`), 220  
`Node` (class in `django_evolution.utils.graph`), 203  
`NODE_TYPE_ANCHOR` (`django_evolution.utils.graph.EvolutionGraph` attribute), 206  
`NODE_TYPE_CREATE_MODEL` (`django_evolution.utils.graph.EvolutionGraph` attribute), 206  
`NODE_TYPE_EVOLUTION` (`django_evolution.utils.graph.EvolutionGraph` attribute), 206  
`NODE_TYPE_MIGRATION` (`django_evolution.utils.graph.EvolutionGraph` attribute), 206  
`NodeNotFoundError`, 203  
`normalize_bool()` (`django_evolution.db.common.BaseEvolutionOperations` method), 179  
`normalize_bool()` (`django_evolution.db.postgresql.EvolutionOperations` method), 183  
`normalize_initial()` (`django_evolution.db.common.BaseEvolutionOperations` method), 179  
`normalize_sql()` (`django_evolution.db.sql_result.SQLResult` method), 185  
`normalize_value()` (`django_evolution.db.common.BaseEvolutionOperations` method), 179  
`NoTransactionSQL` (class in `django_evolution.utils.sql`), 220  
`NullFieldInitialCallback` (class in `django_evolution.placeholders`), 145  
  
**O**  
`objects` (`django_evolution.models.Evolution` attribute), 77  
  
`one_to_many` (`django_evolution.compat.models.GenericForeignKey` attribute), 159  
`one_to_many` (`django_evolution.compat.models.GenericRelation` attribute), 159  
`one_to_one` (`django_evolution.compat.models.GenericForeignKey` attribute), 159  
`one_to_one` (`django_evolution.compat.models.GenericRelation` attribute), 160  
  
`OptionParserWrapper` (class in `django_evolution.compat.commands`), 149  
`OrderedDict` (class in `django_evolution.compat.datastructures`), 150  
  
**P**  
`pickle_dumps()` (in module `django_evolution.compat.py23`), 165  
`pickle_loads()` (in module `django_evolution.compat.py23`), 165  
`placeholder_text` (class in `django_evolution.placeholders.BasePlaceholder` attribute), 145  
`placeholder_text` (class in `django_evolution.placeholders.NullFieldInitialCallback` attribute), 145  
`PlaceholderSerialization` (class in `django_evolution.serialization`), 106  
`pop()` (`django_evolution.compat.datastructures.OrderedDict` method), 151  
`popitem()` (`django_evolution.compat.datastructures.OrderedDict` method), 151  
`prepare()` (`django_evolution.evolve.base.BaseEvolutionTask` method), 64  
`prepare()` (`django_evolution.evolve.evolve_app_task.EvolveAppTask` method), 72  
`prepare()` (`django_evolution.evolve.purge_app_task.PurgeAppTask` method), 63  
`prepare_tasks()` (`django_evolution.evolve.base.BaseEvolutionTask` method), 63  
`prepare_tasks()` (`django_evolution.evolve.evolve_app_task.EvolveAppTask` class method), 70  
`PrimitiveSerialization` (class in `django_evolution.serialization`), 101  
`PROJECT` (`django_evolution.consts.EvolutionsSource` attribute), 57  
`ProjectSignature`, 29  
    project signatures, 29  
`project_sig` (`django_evolution.evolve.Evolver` attribute), 66  
`ProjectSignature` (class in `django_evolution.signature`), 115  
`PurgeAppTask` (class in `django_evolution.evolve.purge_app_task`), 73

**Q**

QSerialization (class in *django\_evolution.serialization*), 107  
queue\_evolv\_all\_apps() (django\_evolution.evolve.evolver.Evolver method), 68  
queue\_evolv\_app() (django\_evolution.evolve.evolver.Evolver method), 68  
queue\_purge\_app() (django\_evolution.evolve.evolver.Evolver method), 68  
queue\_purge\_old\_apps() (django\_evolution.evolve.evolver.Evolver method), 68  
queue\_task() (django\_evolution.evolve.evolver.Evolver method), 68  
QueueEvolverTaskError, 60  
quote\_sql\_param() (django\_evolution.db.common.BaseEvolutionOperations method), 169

**R**

record\_applied\_migrations() (in module django\_evolution.utils.migrations), 214  
register\_global\_custom\_migrations() (in module django\_evolution.utils.migrations), 214  
rel\_class(django\_evolution.compat.models.GenericRelation attribute), 160  
related\_model(django\_evolution.compat.models.GenericForeignKey attribute), 159  
remote\_field(django\_evolution.compat.models.GenericForeignKey attribute), 159  
remove\_app\_sig() (django\_evolution.signature.ProjectSignature method), 116  
remove\_dependencies() (django\_evolution.utils.graph.DependencyGraph method), 205  
remove\_field\_sig() (django\_evolution.signature.ModelSignature method), 123  
remove\_index() (django\_evolution.db.state.DatabaseState method), 193  
remove\_model\_sig() (django\_evolution.signature.AppSignature method), 119  
RemovedInDjangoEvolution30Warning, 58  
RemovedInDjangoEvolution40Warning, 58  
RemovedInNextDjangoEvolutionWarning (in module django\_evolution.deprecation), 59  
rename\_column() (django\_evolution.db.common.BaseEvolutionOperations method), 169  
rename\_column() (django\_evolution.db.mysql.EvolutionOperations method), 180  
rename\_column() (django\_evolution.db.postgresql.EvolutionOperations method), 183  
rename\_column() (django\_evolution.db.sqlite3.EvolutionOperations method), 187

rename\_table() (django\_evolution.db.common.BaseEvolutionOperations method), 169  
rename\_table() (django\_evolution.db.sqlite3.EvolutionOperations method), 187  
RenameAppLabel (built-in class), 21  
RenameAppLabel (class in django\_evolution.mutations.rename\_app\_label), 94  
RenameField (built-in class), 17  
RenameField (class in django\_evolution.mutations.rename\_field), 95  
RenameModel (built-in class), 19  
RenameModel (class in django\_evolution.mutations.rename\_model), 96  
replace\_settings() (django\_evolution.conf.DjangoEvolutionSettings method), 57  
required\_by (django\_evolution.utils.graph.Node attribute), 204  
rescan\_tables() (django\_evolution.db.state.DatabaseState method), 195  
resolve\_related\_fields() (django\_evolution.compat.models.GenericRelation method), 160  
restore\_field\_ref\_constraints() (django\_evolution.db.common.BaseEvolutionOperations method), 178  
run\_checks() (django\_evolution.utils.migrations.MigrationExecutor method), 214  
run\_mutation() (django\_evolution.mutators.app\_mutator.AppMutator method), 140  
run\_mutation() (django\_evolution.mutators.model\_mutator.ModelMutator method), 143  
run\_mutations() (django\_evolution.mutators.app\_mutator.AppMutator method), 140  
run\_simulation() (django\_evolution.mutations.base.BaseMutation method), 83  
run\_sql() (django\_evolution.utils.sql.SQLExecutor method), 221

**S**

serialize() (django\_evolution.signature.AppSignature method), 120  
serialize() (django\_evolution.signature.BaseSignature method), 114  
serialize() (django\_evolution.signature.ConstraintSignature method), 127  
serialize() (django\_evolution.signature.FieldSignature method), 127  
serialize() (django\_evolution.signature.IndexSignature method), 129  
serialize() (django\_evolution.signature.ModelSignature method), 125

```
serialize() (django_evolution.signature.ProjectSignature.serialize_to_signature()
    method), 117                                         (django_evolution.serialization.DictSerialization
serialize_attr() (django_evolution.mutations.base.BaseMutation.serialize_to_signature()
    method), 102                                         class method), 102
serialize_to_python()                                     serialize_to_signature()
    (django_evolution.serialization.BaseSerialization      (django_evolution.serialization.EnumSerialization
    class method), 100                                         class method), 103
serialize_to_python()                                     serialize_to_signature()
    (django_evolution.serialization.ClassSerialization    (django_evolution.serialization.PrimitiveSerialization
    class method), 102                                         class method), 101
serialize_to_python()                                     serialize_to_signature()
    (django_evolution.serialization.CombinedExpressionSerialization
    class method), 107                                         (django_evolution.serialization.QSerialization
serialize_to_python()                                     serialize_to_signature()
    (django_evolution.serialization.DeconstructedSerialization
    class method), 105                                         class method), 107
serialize_to_python()                                     serialize_to_signature()
    (django_evolution.serialization.DictSerialization      (django_evolution.serialization.StringSerialization
    class method), 105                                         class method), 105
serialize_to_python()                                     serialize_to_signature()      (in      module
    (django_evolution.serialization.DictSerialization      django_evolution.serialization), 108
    class method), 102                                         serialize_value() (django_evolution.mutations.base.BaseMutation
                                                               method), 84
serialize_to_python()                                     set_attributes_from_rel()
    (django_evolution.serialization.EnumSerialization     (django_evolution.compat.models.GenericRelation
    class method), 103                                         method), 161
serialize_to_python()                                     set_field_null()
    (django_evolution.serialization.ListSerialization    (django_evolution.db.common.BaseEvolutionOperatio
    class method), 104                                         method), 170
serialize_to_python()                                     set_field_null()
    (django_evolution.serialization.PlaceholderSerialization
    class method), 106                                         (django_evolution.db.mysql.EvolutionOperations
                                                               method), 180
serialize_to_python()                                     set_model_name()      (in      module
    (django_evolution.serialization.PrimitiveSerialization
    class method), 101                                         django_evolution.compat.models), 164
serialize_to_python()                                     setdefault()
    (django_evolution.serialization.QSerialization        (django_evolution.compat.datastructures.OrderedDict
    class method), 107                                         method), 151
serialize_to_python()                                     SetSerialization      (class      in
    (django_evolution.serialization.QSerialization        django_evolution.serialization), 104
    class method), 107                                         setup_fields() (django_evolution.mock_models.MockMeta
                                                               method), 136
serialize_to_python()                                     signature (django_evolution.models.Version attribute),
    (django_evolution.serialization.SetSerialization      76
    class method), 104                                         SignatureField (class in django_evolution.models), 75
serialize_to_python()                                     simulate() (django_evolution.mutations.addField
    (django_evolution.serialization.StringSerialization   AddField
    class method), 105                                         method), 79
serialize_to_python()                                     simulate() (django_evolution.mutations.base.BaseMutation
    (django_evolution.serialization.TupleSerialization   method), 83
    class method), 104                                         simulate() (django_evolution.mutations.change_field.ChangeField
                                                               method), 88
serialize_to_python()                                     simulate() (django_evolution.mutations.change_meta.ChangeMeta
    (in      module      django_evolution.serialization), 108
    class method), 108                                         method), 89
serialize_to_signature()                                simulate() (django_evolution.mutations.delete_application.DeleteApplication
    (django_evolution.serialization.BaseIterableSerialization
    class method), 101                                         method), 90
serialize_to_signature()                                simulate() (django_evolution.mutations.delete_field.DeleteField
    (django_evolution.serialization.BaseSerialization    method), 91
    class method), 99                                         simulate() (django_evolution.mutations.delete_model.DeleteModel
                                                               method), 92
serialize_to_signature()                                simulate() (django_evolution.mutations.move_to_django_migrations.Move
    (django_evolution.serialization.DeconstructedSerialization
    class method), 105                                         method), 93
serialize_to_signature()                                simulate() (django_evolution.mutations.rename_app_label.RenameAppLabel
    (django_evolution.serialization.DeconstructedSerialization
    class method), 105                                         method), 94
```

*method), 94*  
**simulate()** (*django\_evolution.mutations.rename\_field.RenameField*)  
*(in module django\_evolution.mutations.rename\_field), 96*  
**simulate()** (*django\_evolution.mutations.rename\_model.RenameModel*)  
*(django\_evolution.evolve.evolver.Evolver method), 97*  
**simulate()** (*django\_evolution.mutations.sql\_mutation.SQLMutation*)  
*(in module django\_evolution.mutations.sql\_mutation), 98*  
**Simulation** (*class in django\_evolution.mutations.base*),  
*80*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.add\_field.AddField attribute), 78*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.base.BaseMutation attribute), 82*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.change\_field.ChangeField attribute), 87*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.change\_meta.ChangeMeta attribute), 89*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.delete\_application.DeleteApplication attribute), 90*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.delete\_field.DeleteField attribute), 91*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.delete\_model.DeleteModel attribute), 92*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.rename\_field.RenameField attribute), 95*  
**simulation\_failure\_error**  
*(django\_evolution.mutations.rename\_model.RenameModel attribute), 96*  
**SimulationFailure**, 60  
**SortedDict** (*class in django\_evolution.compat.picklers*),  
*164*  
**sql** (*django\_evolution.evolve.base.BaseEvolutionTask* attribute), 63  
**sql** (*django\_evolution.utils.sql.BaseGroupedSQL* attribute), 220  
**sql\_add\_constraints()** (*in module django\_evolution.compat.db*), 155  
**sql\_create\_app()** (*in module django\_evolution.compat.db*), 155  
**sql\_create\_for\_many\_to\_many\_field()** (*in module django\_evolution.compat.db*), 156  
**sql\_create\_models()** (*in module django\_evolution.compat.db*), 156  
**sql\_delete()** (*in module django\_evolution.compat.db*),  
*156*  
**sql\_delete\_constraints()** (*in module django\_evolution.compat.db*),  
*157*  
**sql\_delete\_index()** (*in module django\_evolution.compat.db*), 157  
**sql\_mutate\_for\_field()** (*in module django\_evolution.compat.db*), 157  
**sql\_indexes\_for\_fields()** (*in module django\_evolution.compat.db*), 158  
**sql\_indexes\_for\_model()** (*in module django\_evolution.compat.db*), 158  
**SQLExecutor** (*class in django\_evolution.utils.sql*), 220  
**SQLiteAlterTableSQLResult** (*class in django\_evolution.db.sqlite3*), 186  
**SQLMutation** (*built-in class*), 21  
**SQLMutation** (*class in django\_evolution.mutations.sql\_mutation*),  
*97*  
**SQLMutator** (*class in django\_evolution.mutators.sql\_mutator*),  
*144*  
**SQLResult** (*class in django\_evolution.db.sql\_result*),  
*184*  
**taskify\_field\_ref\_constraints()**  
*(django\_evolution.db.common.BaseEvolutionOperations method), 178*  
**StringSerialization** (*class in django\_evolution.serialization*), 105  
**Supported\_change\_attrs**  
*(django\_evolution.db.common.BaseEvolutionOperations attribute), 166*  
**Supported\_change\_meta**  
*(django\_evolution.db.common.BaseEvolutionOperations attribute), 166*  
**Supported\_change\_meta**  
*(django\_evolution.db.sqlite3.EvolutionOperations attribute), 186*  
**supports\_constraints** (*in module django\_evolution.support*), 147  
**supports\_db\_table\_comments** (*in module django\_evolution.support*), 147  
**supports\_f\_comparison** (*in module django\_evolution.support*), 147  
**supports\_index\_feature()** (*in module django\_evolution.support*), 147  
**supports\_index\_together** (*in module django\_evolution.support*), 146  
**supports\_indexes** (*in module django\_evolution.support*), 146  
**supports\_migrations** (*in module django\_evolution.support*), 147  
**supports\_q\_comparison** (*in module django\_evolution.support*), 146

### T

tasks (*django\_evolution.evolve.evolver.Evolver* property), 67  
to\_python() (*django\_evolution.models.SignatureField* method), 75  
to\_sql() (*django\_evolution.db.sql\_result.AlterTableSQLResult* method), 185  
to\_sql() (*django\_evolution.db.sql\_result.SQLResult* method), 185  
to\_sql() (*django\_evolution.db.sqlite3.SQLiteAlterTableSQLResult* method), 186  
to\_sql() (*django\_evolution.mutators.app\_mutator.AppMutator* method), 140  
to\_sql() (*django\_evolution.mutators.model\_mutator.ModelMutator* method), 143  
to\_sql() (*django\_evolution.mutators.sql\_mutator.SQLMutator* method), 144  
to\_targets() (*django\_evolution.utils.migrations.MigrationList* method), 211  
transaction() (*django\_evolution.evolve.evolver.Evolver* method), 69  
truncate\_name() (*in module django\_evolution.compat.db*), 158  
TupleSerialization (*class in django\_evolution.serialization*), 104

### U

unique\_together (*django\_evolution.signature.ModelSignature* property), 122  
unrecord\_applied\_migrations() (*in module django\_evolution.utils.migrations*), 215  
update() (*django\_evolution.compat.datastructures.OrderedDict* method), 151  
update() (*django\_evolution.utils.migrations.MigrationList* method), 210  
UpgradeMethod (*class in django\_evolution.consts*), 57  
use\_argparse (*django\_evolution.compat.commands.BaseCommand* property), 149

### V

validate\_sig\_version() (*in module django\_evolution.signature*), 132  
value\_to\_string() (*django\_evolution.compat.models.GenericRelation* method), 161  
value\_to\_string() (*django\_evolution.models.SignatureField* method), 75  
values() (*django\_evolution.compat.datastructures.OrderedDict* method), 151  
verbosity (*django\_evolution.evolve.evolver.Evolver* attribute), 66  
Version (*class in django\_evolution.models*), 76  
version (*django\_evolution.evolve.evolver.Evolver* attribute), 66

version (*django\_evolution.models.Evolution* attribute), 77

version\_id (*django\_evolution.models.Evolution* attribute), 77

VersionManager (*class in django\_evolution.models*), 74

### W

walk\_model\_tree() (*in module django\_evolution.utils.models*), 218

warn() (*django\_evolution.deprecation.BaseRemovedInDjangoEvolutionWarning* class method), 58

wipe\_evolution command line option

when (*django\_evolution.models.Version* attribute), 76

--noinput, 27

EVOLUTION\_LABEL, 27