Migrations

Migrations are Django’s way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They’re designed to be mostly automatic, but you’ll need to know when to make migrations, when to run them, and the common problems you might run into.

The Commands

There are several commands which you will use to interact with migrations and Django’s handling of database schema:

- `migrate`, which is responsible for applying and unapplying migrations.
- `makemigrations`, which is responsible for creating new migrations based on the changes you have made to your models.
- `sqlmigrate`, which displays the SQL statements for a migration.
- `showmigrations`, which lists a project’s migrations and their status.

You should think of migrations as a version control system for your database schema. `makemigrations` is responsible for packaging up your model changes into individual migration files - analogous to commits - and `migrate` is responsible for applying those to your database.

The migration files for each app live in a “migrations” directory inside of that app, and are designed to be committed to, and distributed as part of, its codebase. You should be making them once on your development machine and then running the same migrations on your colleagues’ machines, your staging machines, and eventually your production machines.

Note: It is possible to override the name of the package which contains the migrations on a per-app basis by modifying the `MIGRATION_MODULES` setting.

Migrations will run the same way on the same dataset and produce consistent results, meaning that what you see in development and staging is, under the same circumstances, exactly what will happen in production.

Django will make migrations for any change to your models or fields - even options that don’t affect the database - as the only way it can reconstruct a field correctly is to have all the changes in the history, and you might need those options in some data migrations later on (for example, if you’ve set custom validators).

Backend Support

Migrations are supported on all backends that Django ships with, as well as any third-party backends if they have programmed in support for schema alteration (done via the SchemaEditor class).

However, some databases are more capable than others when it comes to schema migrations; some of the caveats are covered below.

PostgreSQL

PostgreSQL is the most capable of all the databases here in terms of schema support.

MySQL

MySQL lacks support for transactions around schema alteration operations, meaning that if a migration fails to apply you will have to manually unpick the changes in order to try again (it's impossible to roll back to an earlier point).

MySQL 8.0 introduced significant performance enhancements for DDL operations, making them more efficient and reducing the need for full table rebuilds. However, it cannot guarantee a complete absence of locks or interruptions. In situations where locks are still necessary, the duration of these operations will be proportionate to the number of rows involved.

Finally, MySQL has a relatively small limit on the combined size of all columns an index covers. This means that indexes that are possible on other backends will fail to be created under MySQL.

SQLite

SQLite has very little built-in schema alteration support, and so Django attempts to emulate it by:

- Creating a new table with the new schema
- Copying the data across
- Dropping the old table
- Renaming the new table to match the original name

This process generally works well, but it can be slow and occasionally buggy. It is not recommended that you run and migrate SQLite in a production environment unless you are very aware of the risks and its limitations; the support Django ships with is designed to allow developers to use SQLite on their local machines to develop less complex Django projects without the need for a full database.

Workflow

Django can create migrations for you. Make changes to your models - say, add a field and remove a model - and then run `makemigrations`:

```
$ python manage.py makemigrations
Migrations for 'books':
  books/migrations/0003_auto.py:
    ~ Alter field author on book
```

Your models will be scanned and compared to the versions currently contained in your migration files, and then a new set of migrations will be written out. Make sure to read the output to see what `makemigrations` thinks you have changed - it's not perfect, and for complex changes it might not be detecting what you expect.

Once you have your new migration files, you should apply them to your database to make sure they work as expected:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: books
Running migrations:
  Rendering model states... DONE
  Applying books.0003_auto... OK
```

Once the migration is applied, commit the migration and the models change to your version control system as a single commit - that way, when other developers (or your production servers) check out the code, they'll get both the changes to your models and the accompanying migration at the same time.

If you want to give the migration(s) a meaningful name instead of a generated one, you can use the `makemigrations --name` option:

```
$ python manage.py makemigrations --name changed_my_model your_app_label
```

Version control

Because migrations are stored in version control, you'll occasionally come across situations where you and another developer have both committed a migration to the same app at the same time, resulting in two migrations with the same number.

Don't worry - the numbers are just there for developers' reference, Django just cares that each migration has a different name. Migrations specify which other migrations they depend on - including earlier migrations in the same app - in the file, so it's possible to detect when there's two new migrations for the same app that aren't ordered.

When this happens, Django will prompt you and give you some options. If it thinks it's safe enough, it will offer to automatically linearize the two migrations for you. If not, you'll have to go in and modify the migrations yourself - don't worry, this isn't difficult, and is explained more in Migration files below.

Transactions

On databases that support DDL transactions (SQLite and PostgreSQL), all migration operations will run inside a single transaction by default. In contrast, if a database doesn't support DDL transactions (e.g. MySQL, Oracle) then all operations will run without a transaction.

You can prevent a migration from running in a transaction by setting the `atomic` attribute to `False`. For example:

```
from django.db import migrations


class Migration(migrations.Migration):
    atomic = False
```

It's also possible to execute parts of the migration inside a transaction using `atomic()` or by passing `atomic=True` to `RunPython`. See Non-atomic migrations for more details.

Dependencies

While migrations are per-app, the tables and relationships implied by your models are too complex to be created for one app at a time. When you make a migration that requires something else to run - for example, you add a `ForeignKey` in your `books` app to your `authors` app - the resulting migration will contain a dependency on a migration in `authors`.

This means that when you run the migrations, the `authors` migration runs first and creates the table the `ForeignKey` references, and then the migration that makes the `ForeignKey` column runs afterward and creates the constraint. If this didn't happen, the migration would try to create the `ForeignKey` column without the table it's referencing existing and your database would throw an error.

This dependency behavior affects most migration operations where you restrict to a single app. Restricting to a single app (either in `makemigrations` or `migrate`) is a best-efforts promise, and not a guarantee; any other apps that need to be used to get dependencies correct will be.

Apps without migrations must not have relations (`ForeignKey`, `ManyToManyField`, etc.) to apps with migrations. Sometimes it may work, but it's not supported.

Swappable dependencies

```
django.db.migrations.swappable_dependency(value)
```

The `swappable_dependency()` function is used in migrations to declare "swappable" dependencies on migrations in the app of the swapped-in model, currently, on the first migration of this app. As a consequence, the swapped-in model should be created in the initial migration. The argument `value` is a string `"<app label>.<model>"` describing an app label and a model name, e.g. `"myapp.MyModel"`.

By using `swappable_dependency()`, you inform the migration framework that the migration relies on another migration which sets up a swappable model, allowing for the possibility of substituting the model with a different implementation in the future. This is typically used for referencing models that are subject to customization or replacement, such as the custom user model (`settings.AUTH_USER_MODEL`, which defaults to `"auth.User"`) in Django's authentication system.

Migration files

Migrations are stored as an on-disk format, referred to here as "migration files". These files are actually normal Python files with an agreed-upon object layout, written in a declarative style.

A basic migration file looks like this:

```
from django.db import migrations, models


class Migration(migrations.Migration):
    dependencies = [("migrations", "0001_initial")]

    operations = [
        migrations.DeleteModel("Tribble"),
        migrations.AddField("Author", "rating", models.IntegerField(default=0)),
    ]
```

What Django looks for when it loads a migration file (as a Python module) is a subclass of `django.db.migrations.Migration` called `Migration` . It then inspects this object for four attributes, only two of which are used most of the time:

- `dependencies` , a list of migrations this one depends on.
- `operations` , a list of `Operation` classes that define what this migration does.

The operations are the key; they are a set of declarative instructions which tell Django what schema changes need to be made. Django scans them and builds an in-memory representation of all of the schema changes to all apps, and uses this to generate the SQL which makes the schema changes.

That in-memory structure is also used to work out what the differences are between your models and the current state of your migrations; Django runs through all the changes, in order, on an in-memory set of models to come up with the state of your models last time you ran `makemigrations` . It then uses these models to compare against the ones in your `models.py` files to work out what you have changed.

You should rarely, if ever, need to edit migration files by hand, but it's entirely possible to write them manually if you need to. Some of the more complex operations are not autodetectable and are only available via a hand-written migration, so don't be scared about editing them if you have to.

Custom fields

You can't modify the number of positional arguments in an already migrated custom field without raising a `TypeError` . The old migration will call the modified `__init__` method with the old signature. So if you need a new argument, please create a keyword argument and add something like `assert 'argument_name' in kwargs` in the constructor.

Model managers

You can optionally serialize managers into migrations and have them available in `RunPython` operations. This is done by defining a `use_in_migrations` attribute on the manager class:

```
class MyManager(models.Manager):
    use_in_migrations = True


class MyModel(models.Model):
    objects = MyManager()
```

If you are using the `from_queryset()` function to dynamically generate a manager class, you need to inherit from the generated class to make it importable:

```
class MyManager(MyBaseManager.from_queryset(CustomQuerySet)):
    use_in_migrations = True


class MyModel(models.Model):
    objects = MyManager()
```

Please refer to the notes about Historical models in migrations to see the implications that come along.

Initial migrations

```
Migration.initial
```

The "initial migrations" for an app are the migrations that create the first version of that app's tables. Usually an app will have one initial migration, but in some cases of complex model interdependencies it may have two or more.

Initial migrations are marked with an `initial = True` class attribute on the migration class. If an `initial` class attribute isn't found, a migration will be considered "initial" if it is the first migration in the app (i.e. if it has no dependencies on any other migration in the same app).

When the `migrate --fake-initial` option is used, these initial migrations are treated specially. For an initial migration that creates one or more tables (`CreateModel` operation), Django checks that all of those tables already exist in the database and fake-applies the migration if so. Similarly, for an initial migration that adds one or more fields (`AddField` operation), Django checks that all of the respective columns already exist in the database and fake-applies the migration if so. Without `--fake-initial` , initial migrations are treated no differently from any other migration.

History consistency

As previously discussed, you may need to linearize migrations manually when two development branches are joined. While editing migration dependencies, you can inadvertently create an inconsistent history state where a migration has been applied but some of its dependencies haven't. This is a strong indication that the dependencies are incorrect, so Django will refuse to run migrations or make new migrations until it's fixed. When using multiple databases, you can use the `allow_migrate()` method of database routers to control which databases `makemigrations` checks for consistent history.

Adding migrations to apps

New apps come preconfigured to accept migrations, and so you can add migrations by running `makemigrations` once you've made some changes.

If your app already has models and database tables, and doesn't have migrations yet (for example, you created it against a previous Django version), you'll need to convert it to use migrations by running:

```
$ python manage.py makemigrations your_app_label
```

This will make a new initial migration for your app. Now, run `python manage.py migrate --fake-initial`, and Django will detect that you have an initial migration *and* that the tables it wants to create already exist, and will mark the migration as already applied. (Without the `migrate --fake-initial` flag, the command would error out because the tables it wants to create already exist.)

Note that this only works given two things:

- You have not changed your models since you made their tables. For migrations to work, you must make the initial migration *first* and then make changes, as Django compares changes against migration files, not the database.
- You have not manually edited your database - Django won't be able to detect that your database doesn't match your models, you'll just get errors when migrations try to modify those tables.

Reversing migrations

Migrations can be reversed with `migrate` by passing the number of the previous migration. For example, to reverse migration `books.0003`:

```
$ python manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto... OK
```

```
...\> py manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto... OK
```

If you want to reverse all migrations applied for an app, use the name `zero`:

```
$ python manage.py migrate books zero
Operations to perform:
  Unapply all migrations: books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0002_auto... OK
  Unapplying books.0001_initial... OK
```

```
...\> py manage.py migrate books zero
Operations to perform:
  Unapply all migrations: books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0002_auto... OK
  Unapplying books.0001_initial... OK
```

A migration is irreversible if it contains any irreversible operations. Attempting to reverse such migrations will raise `IrreversibleError`:

```
$ python manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto...Traceback (most recent call last):
django.db.migrations.exceptions.IrreversibleError: Operation <RunSQL  sql='DROP TABLE demo_books'> in
books.0003_auto is not reversible
```

```
...\> py manage.py migrate books 0002
Operations to perform:
  Target specific migration: 0002_auto, from books
Running migrations:
  Rendering model states... DONE
  Unapplying books.0003_auto...Traceback (most recent call last):
django.db.migrations.exceptions.IrreversibleError: Operation <RunSQL  sql='DROP TABLE demo_books'> in
books.0003_auto is not reversible
```

Historical models

When you run migrations, Django is working from historical versions of your models stored in the migration files. If you write Python code using the `RunPython` operation, or if you have `allow_migrate` methods on your database routers, you **need to use** these historical model versions rather than importing them directly.

Warning: If you import models directly rather than using the historical models, your migrations *may work initially* but will fail in the future when you try to rerun old migrations (commonly, when you set up a new installation and run through all the migrations to set up the database).

This means that historical model problems may not be immediately obvious. If you run into this kind of failure, it’s OK to edit the migration to use the historical models rather than direct imports and commit those changes.

Because it’s impossible to serialize arbitrary Python code, these historical models will not have any custom methods that you have defined. They will, however, have the same fields, relationships, managers (limited to those with `use_in_migrations = True`) and `Meta` options (also versioned, so they may be different from your current ones).

Warning: This means that you will NOT have custom `save()` methods called on objects when you access them in migrations, and you will NOT have any custom constructors or instance methods. Plan appropriately!

References to functions in field options such as `upload_to` and `limit_choices_to` and model manager declarations with managers having `use_in_migrations = True` are serialized in migrations, so the functions and classes will need to be kept around for as long as there is a migration referencing them. Any custom model fields will also need to be kept, since these are imported directly by migrations.

In addition, the concrete base classes of the model are stored as pointers, so you must always keep base classes around for as long as there is a migration that contains a reference to them. On the plus side, methods and managers from these base classes inherit normally, so if you absolutely need access to these you can opt to move them into a superclass.

To remove old references, you can squash migrations or, if there aren’t many references, copy them into the migration files.

Considerations when removing model fields

Similar to the “references to historical functions” considerations described in the previous section, removing custom model fields from your project or third-party app will cause a problem if they are referenced in old migrations.

To help with this situation, Django provides some model field attributes to assist with model field deprecation using the system checks framework.

Add the `system_check_deprecated_details` attribute to your model field similar to the following:

```python
class IPAddressField(Field):
    system_check_deprecated_details = {
        "msg": (
            "IPAddressField has been deprecated. Support for it (except "
            "in historical migrations) will be removed in Django 1.9."
        ),
        "hint": "Use GenericIPAddressField instead.",  # optional
        "id": "fields.W900",  # pick a unique ID for your field.
    }
```

After a deprecation period of your choosing (two or three feature releases for fields in Django itself), change the `system_check_deprecated_details` attribute to `system_check_removed_details` and update the dictionary similar to:

```
class IPAddressField(Field):
    system_check_removed_details = {
        "msg": (
            "IPAddressField has been removed except for support in "
            "historical migrations."
        ),
        "hint": "Use GenericIPAddressField instead.",
        "id": "fields.E900",  # pick a unique ID for your field.
    }
```

You should keep the field's methods that are required for it to operate in database migrations such as `__init__()`, `deconstruct()`, and `get_internal_type()`. Keep this stub field for as long as any migrations which reference the field exist. For example, after squashing migrations and removing the old ones, you should be able to remove the field completely.

Data Migrations

As well as changing the database schema, you can also use migrations to change the data in the database itself, in conjunction with the schema if you want.

Migrations that alter data are usually called "data migrations"; they're best written as separate migrations, sitting alongside your schema migrations.

Django can't automatically generate data migrations for you, as it does with schema migrations, but it's not very hard to write them. Migration files in Django are made up of Operations, and the main operation you use for data migrations is `RunPython`.

To start, make an empty migration file you can work from (Django will put the file in the right place, suggest a name, and add dependencies for you):

```
python manage.py makemigrations --empty yourappname
```

Then, open up the file; it should look something like this:

```
# Generated by Django A.B on YYYY-MM-DD HH:MM
from django.db import migrations


class Migration(migrations.Migration):
    dependencies = [
        ("yourappname", "0001_initial"),
    ]

    operations = []
```

Now, all you need to do is create a new function and have `RunPython` use it. `RunPython` expects a callable as its argument which takes two arguments - the first is an app registry that has the historical versions of all your models loaded into it to match where in your history the migration sits, and the second is a SchemaEditor, which you can use to manually effect database schema changes (but beware, doing this can confuse the migration autodetector!)

Let's write a migration that populates our new `name` field with the combined values of `first_name` and `last_name` (we've come to our senses and realized that not everyone has first and last names). All we need to do is use the historical model and iterate over the rows:

```
from django.db import migrations


def combine_names(apps, schema_editor):
    # We can't import the Person model directly as it may be a newer
    # version than this migration expects. We use the historical version.
    Person = apps.get_model("yourappname", "Person")
    for person in Person.objects.all():
        person.name = f"{person.first_name} {person.last_name}"
        person.save()


class Migration(migrations.Migration):
    dependencies = [
        ("yourappname", "0001_initial"),
    ]

    operations = [
        migrations.RunPython(combine_names),
    ]
```

Once that's done, we can run `python manage.py migrate` as normal and the data migration will run in place alongside other migrations.

You can pass a second callable to `RunPython` to run whatever logic you want executed when migrating backwards. If this callable is omitted, migrating backwards will raise an exception.

Accessing models from other apps

When writing a `RunPython` function that uses models from apps other than the one in which the migration is located, the migration's `dependencies` attribute should include the latest migration of each app that is involved, otherwise you may get an error similar to: `LookupError: No installed app with label 'myappname'` when you try to retrieve the model in the `RunPython` function using `apps.get_model()`.

In the following example, we have a migration in `app1` which needs to use models in `app2`. We aren't concerned with the details of `move_m1` other than the fact it will need to access models from both apps. Therefore we've added a dependency that specifies the last migration of `app2`:

```python
class Migration(migrations.Migration):
    dependencies = [
        ("app1", "0001_initial"),
        # added dependency to enable using models from app2 in move_m1
        ("app2", "0004_foobar"),
    ]

    operations = [
        migrations.RunPython(move_m1),
    ]
```

More advanced migrations

If you're interested in the more advanced migration operations, or want to be able to write your own, see the migration operations reference and the "how-to" on writing migrations.

Squashing migrations

You are encouraged to make migrations freely and not worry about how many you have; the migration code is optimized to deal with hundreds at a time without much slowdown. However, eventually you will want to move back from having several hundred migrations to just a few, and that's where squashing comes in.

Squashing is the act of reducing an existing set of many migrations down to one (or sometimes a few) migrations which still represent the same changes.

Django does this by taking all of your existing migrations, extracting their `Operation` s and putting them all in sequence, and then running an optimizer over them to try and reduce the length of the list - for example, it knows that `CreateModel` and `DeleteModel` cancel each other out, and it knows that `AddField` can be rolled into `CreateModel`.

Once the operation sequence has been reduced as much as possible - the amount possible depends on how closely intertwined your models are and if you have any `RunSQL` or `RunPython` operations (which can't be optimized through unless they are marked as `elidable`) - Django will then write it back out into a new set of migration files.

These files are marked to say they replace the previously-squashed migrations, so they can coexist with the old migration files, and Django will intelligently switch between them depending where you are in the history. If you're still part-way through the set of migrations that you squashed, it will keep using them until it hits the end and then switch to the squashed history, while new installs will use the new squashed migration and skip all the old ones.

This enables you to squash and not mess up systems currently in production that aren't fully up-to-date yet. The recommended process is to squash, keeping the old files, commit and release, wait until all systems are upgraded with the new release (or if you're a third-party project, ensure your users upgrade releases in order without skipping any), and then remove the old files, commit and do a second release.

The command that backs all this is `squashmigrations` - pass it the app label and migration name you want to squash up to, and it'll get to work:

```
$ ./manage.py squashmigrations myapp 0004
Will squash the following migrations:
 - 0001_initial
 - 0002_some_change
 - 0003_another_change
 - 0004_undo_something
Do you wish to proceed? [y/N] y
Optimizing...
  Optimized from 12 operations to 7 operations.
Created new squashed migration
/home/andrew/Programs/DjangoTest/test/migrations/0001_squashed_0004_undo_something.py
  You should commit this migration but leave the old ones in place;
  the new migration will be used for new installs. Once you are sure
  all instances of the codebase have applied the migrations you squashed,
  you can delete them.
```

Use the `squashmigrations --squashed-name` option if you want to set the name of the squashed migration rather than use an autogenerated one.

Note that model interdependencies in Django can get very complex, and squashing may result in migrations that do not run; either mis-optimized (in which case you can try again with `--no-optimize`, though you should also report an issue), or with a `CircularDependencyError`, in which case you can manually resolve it.

To manually resolve a `CircularDependencyError`, break out one of the ForeignKeys in the circular dependency loop into a separate migration, and move the dependency on the other app with it. If you're unsure, see how `makemigrations` deals with the problem when asked to create brand new migrations from your models. In a future release of Django, `squashmigrations` will be updated to attempt to resolve these errors itself.

Once you've squashed your migration, you should then commit it alongside the migrations it replaces and distribute this change to all running instances of your application, making sure that they run `migrate` to store the change in their database.

You must then transition the squashed migration to a normal migration by:

- Deleting all the migration files it replaces.
- Updating all migrations that depend on the deleted migrations to depend on the squashed migration instead.
- Removing the `replaces` attribute in the `Migration` class of the squashed migration (this is how Django tells that it is a squashed migration).

Note: Once you've squashed a migration, you should not then re-squash that squashed migration until you have fully transitioned it to a normal migration.

Pruning references to deleted migrations: If it is likely that you may reuse the name of a deleted migration in the future, you should remove references to it from Django's migrations table with the `migrate --prune` option.

Serializing values

Migrations are Python files containing the old definitions of your models - thus, to write them, Django must take the current state of your models and serialize them out into a file.

While Django can serialize most things, there are some things that we just can't serialize out into a valid Python representation - there's no Python standard for how a value can be turned back into code (`repr()` only works for basic values, and doesn't specify import paths).

Django can serialize the following:

- `int` , `float` , `bool` , `str` , `bytes` , `None` , `NoneType`
- `list` , `set` , `tuple` , `dict` , `range` .
- `datetime.date` , `datetime.time` , and `datetime.datetime` instances (include those that are timezone-aware)
- `decimal.Decimal` instances
- `enum.Enum` and `enum.Flag` instances
- `uuid.UUID` instances
- `functools.partial()` and `functools.partialmethod` instances which have serializable `func` , `args` , and `keywords` values.
- Pure and concrete path objects from `pathlib`. Concrete paths are converted to their pure path equivalent, e.g. `pathlib.PosixPath` to `pathlib.PurePosixPath`.
- `os.PathLike` instances, e.g. `os.DirEntry`, which are converted to `str` or `bytes` using `os.fspath()`.
- `LazyObject` instances which wrap a serializable value.
- Enumeration types (e.g. `TextChoices` or `IntegerChoices`) instances.
- Any Django field
- Any function or method reference (e.g. `datetime.datetime.today`) (must be in module's top-level scope)
  - Functions may be decorated if wrapped properly, i.e. using `functools.wraps()`
  - The `functools.cache()` and `functools.lru_cache()` decorators are explicitly supported
- Unbound methods used from within the class body
- Any class reference (must be in module's top-level scope)
- Anything with a custom `deconstruct()` method (see below)

Changed in Django 5.0:
Serialization support for functions decorated with `functools.cache()` or `functools.lru_cache()` was added.

Django cannot serialize:

- Nested classes
- Arbitrary class instances (e.g. `MyClass(4.3, 5.7)`)
- Lambdas

Custom serializers

You can serialize other types by writing a custom serializer. For example, if Django didn't serialize `Decimal` by default, you could do this:

```
from decimal import Decimal

from django.db.migrations.serializer import BaseSerializer
from django.db.migrations.writer import MigrationWriter


class DecimalSerializer(BaseSerializer):
    def serialize(self):
        return repr(self.value), {"from decimal import Decimal"}


MigrationWriter.register_serializer(Decimal, DecimalSerializer)
```

The first argument of `MigrationWriter.register_serializer()` is a type or iterable of types that should use the serializer.

The `serialize()` method of your serializer must return a string of how the value should appear in migrations and a set of any imports that are needed in the migration.

Adding a `deconstruct()` method

You can let Django serialize your own custom class instances by giving the class a `deconstruct()` method. It takes no arguments, and should return a tuple of three things `(path, args, kwargs)` :

- `path` should be the Python path to the class, with the class name included as the last part (for example, `myapp.custom_things.MyClass`). If your class is not available at the top level of a module it is not serializable.
- `args` should be a list of positional arguments to pass to your class' `__init__` method. Everything in this list should itself be serializable.
- `kwargs` should be a dict of keyword arguments to pass to your class' `__init__` method. Every value should itself be serializable.

Note: This return value is different from the `deconstruct()` method for custom fields which returns a tuple of four items.

Django will write out the value as an instantiation of your class with the given arguments, similar to the way it writes out references to Django fields.

To prevent a new migration from being created each time `makemigrations` is run, you should also add a `__eq__()` method to the decorated class. This function will be called by Django's migration framework to detect changes between states.

As long as all of the arguments to your class' constructor are themselves serializable, you can use the `@deconstructible` class decorator from `django.utils.deconstruct` to add the `deconstruct()` method:

```python
from django.utils.deconstruct import deconstructible


@deconstructible
class MyCustomClass:
    def __init__(self, foo=1):
        self.foo = foo
        ...

    def __eq__(self, other):
        return self.foo == other.foo
```

The decorator adds logic to capture and preserve the arguments on their way into your constructor, and then returns those arguments exactly when deconstruct() is called.

Supporting multiple Django versions

If you are the maintainer of a third-party app with models, you may need to ship migrations that support multiple Django versions. In this case, you should always run `makemigrations` **with the lowest Django version you wish to support**.

The migrations system will maintain backwards-compatibility according to the same policy as the rest of Django, so migration files generated on Django X.Y should run unchanged on Django X.Y+1. The migrations system does not promise forwards-compatibility, however. New features may be added, and migration files generated with newer versions of Django may not work on older versions.

See also:

The Migrations Operations Reference

Covers the schema operations API, special operations, and writing your own operations.

The Writing Migrations "how-to"

Explains how to structure and write database migrations for different scenarios you might encounter.