South Documentation

Release 0.7

Andrew Godwin

June 09, 2012

CONTENTS

South is a tool to provide consistent, easy-to-use and database-agnostic migrations for Django applications.

This is the documentation for the current version series (0.7.x); previous versions' documentation was written directly into our wiki, but is mostly a subset of what is written here.

If you want to view the old docs (for example, if you need something that was only in 0.6) they're still available.

CHAPTER ONE

SUPPORT

For initial help with problems, see our mailing list, or #django-south on freenode. If you find a real bug, then file a new ticket.

CHAPTER

TWO

CONTENTS

2.1 About South

South brings migrations to Django applications. Its main objectives are to provide a simple, stable and database-independent migration layer to prevent all the hassle schema changes over time bring to your Django applications.

We try to make South both as easy-to-use and intuitive as possible, by making it automate most of your schema-changing tasks, while at the same time providing a powerful set of tools for large or complex projects; you can easily write your own migrations by hand, or even use the database altering API directly.

While South started as a relative unknown in the Django database-schema-altering world, it has slowly risen in popularity and is now widely regarded as the most popular schema migration tool for Django.

2.1.1 Key features

South has a few key features:

- Automatic migration creation: South can see what's changed in your models.py file and automatically write migrations that match your changes.
- Database independence: As far as possible, South is completely database-agnostic, supporting five different database backends.
- App-savvy: South knows and works with the concept of Django apps, allowing you to use migrations for some of your apps and leave the rest to carry on using syncdb.
- VCS-proof: South will notice if someone else commits migrations to the same app as you and they conflict.

2.1.2 A brief history

South was originally developed at Torchbox in 2008, when no existing solution provided the workflow and features that were needed. It was open-sourced shortly thereafter, and quickly gained steam after the Schema Evolution panel at DjangoCon 2008.

Sometime in 2009, it became the most popular of the various migration alternatives, and seems to have been going strong ever since. While there have been growing calls to integrate South, or something like it, into Django itself, such an integration has not yet been made, mostly due to the relative immaturity of database migration solutions.

2.2 What are migrations?

For the uninitiated, migrations (also known as 'schema evolution' or 'mutations') are a way of changing your database schema from one version into another. Django by itself can only do this by adding new models, but nearly all projects will find themselves changing other aspects of models - be it adding a new field to a model, or changing a database column to have null=True.

South, and other solutions, provide a way of getting round this by giving you the tools to easily and predictably upgrade your database schema. You write migrations, which tell South how to upgrade from one version to the next, and by stringing these migrations together you can move forwards (or backwards) through the history of your database schema.

In South, the migrations also form the way of creating your database initially - the first migration simply migrates from an empty schema to your first tables. This way, running through all the migrations brings your database up-to-date with the most current version of the app, and if you already have an older version, you simply need to run through the ones that appeared since last time.

Running through the *tutorial* will give you a good idea of how migrations work and how they're useful to you, with some solid examples.

2.3 Installation

South's current release is *0.7.5*.

There are a few different ways to install South:

- *Using easy_install* (or pip), which is recommended if you want stable releases.
- *Using a Mercurial checkout*, recommended if you want cutting-edge features.
- *Using our downloadable archives*, useful if you don't have easy_install or Mercurial.

Some Linux distributions are also starting to include South in their package repositories; if you're running unstable Debian you can `apt-get install python-django-south`, and on new Fedoras you can use `yum install Django-south`. Note that this may give you an older version - check the version before using the packages.

South should work with versions of Django from 0.97-pre through to 1.2, although some features (such as multi-db) may not be available for older Django versions.

2.3.1 Using easy_install

If you have easy_install available on your system, just type:

```
easy_install South
```

If you've already got an old version of South, and want to upgrade, use:

```
easy_install -U South
```

That's all that's needed to install the package; you'll now want to *configure your Django installation*.

2.3.2 Using Mercurial

You can install directly from our Mercurial repo, allowing you to recieve updates and bugfixes as soon as they're made. You'll need Mercurial installed on your system; if it's not already, you'll want to get it. The package name is `mercurial` on most Linux distributions; OSX and Windows users can download packages from http://mercurial.berkwood.com.

Make sure you're in a directory where you want the `south` directory to appear, and run:

```
hg clone http://bitbucket.org/andrewgodwin/south/
```

To update an existing Mercurial checkout to the newest version, run:

```
hg pull
hg up -C tip
```

(Rather than running from tip, you can also use the `stableish` tag, which is manually set on reasonably stable trunk commits, or pick a version number tag.)

Once you have this directory, move onto *Installing from a directory*.

2.3.3 Using downloadable archives

If you're averse to using Mercurial, and don't have easy_install available, then you can install from one of our `.tar.gz` files.

First, download the archive of your choice from our releases page, and extract it to create a `south` folder. Then, proceed with our instructions for *Installing from a directory*.

2.3.4 Installing from a directory

If you've obtained a copy of South using either Mercurial or a downloadable archive, you'll need to install the copy you have system-wide. Try running:

```
python setup.py develop
```

If that fails, you don't have `setuptools` or an equivalent installed; either install them, or run:

```
python setup.py install
```

Note that `develop` sets the installed version to run from the directory you just created, while `install` copies all the files to Python's `site-packages` folder, meaning that if you update your checkout you'll need to re-run `install`.

You could also install South locally for only one project, by either including with your project and modifying `sys.path` in your settings file, or (preferably) by using virtualenv, pip and a requirements.txt. A tutorial in how to use these is outside the scope of this documentation, but there are tutorials elsewhere.

Once you've done one of those, you'll want to *configure your Django installation*.

2.3.5 Configuring your Django installation

Now you've installed South system-wide, you'll need to configure Django to use it. Doing so is simple; just edit your `settings.py` and add `'south'` to the end of `INSTALLED_APPS`.

If Django doesn't seem to pick this up, check that you're not overriding `INSTALLED_APPS` elsewhere, and that you can run `import south` from inside `./manage.py shell` with no errors.

Once South is added in, you'll need to run `./manage.py syncdb` to make the South migration-tracking tables (South doesn't use migrations for its own models, for various reasons).

Now South is loaded into your project and ready to go, you'll probably want to take a look at our *Tutorial*.

2.4 Tutorial

This is South's new tutorial, designed to give a reasonably comprehensive grounding in all of the basic features of South. We recommend you work through from the beginning if you're new, or dip in if you already know about some of the features.

Advanced features have their own extensive documentation, as well; the *main page* gives you an overview of all the topics.

2.4.1 Part 1: The Basics

Welcome to the South tutorial; here, we'll try and cover all the basic usage of South, as well as giving you some general hints about what else to do.

If you've never heard of the idea of a migrations library, then please read *What are migrations?* first; that will help you get a better understanding of what both South (and others, such as django-evolution) are trying to achieve.

This tutorial assumes you have South installed correctly; if not, see the *installation instructions*.

Starting off

In this tutorial, we'll follow the process of using migrations on a brand new app. Don't worry about converting your existing apps; we'll cover that in the next part.

The first thing to note is that South is per-application; migrations are stored along with the app's code [1]. If an app doesn't have any migrations defined, South will ignore it, and it will behave as normal (that is, using syncdb).

So, find a project to work in (or make a new one, and set it up with a database and other settings), and let's create our new app:

```
./manage.py startapp southtut
```

As usual, this should make a new directory `southtut/`. First, add it to `INSTALLED_APPS`, then open up the newly-created `southtut/models.py`, and create a new model:

```python
from django.db import models

class Knight(models.Model):
    name = models.CharField(max_length=100)
    of_the_round_table = models.BooleanField()
```

It's quite simple, but it'll do. Now, instead of running `syncdb` to create a table for the model in our database, we'll create a migration for it.

The First Migration

South has several ways of creating migrations; some are automatic, some are manual. As a basic user, you'll probably use the two automatic ways - `--auto` and `--initial`.

`--auto` looks at the previous migration, works out what's changed, and creates a migration which applies the differences - for example, if you add a field to a model, `--auto` will notice this, and make a migration which creates a new column for that field on its model's table.

[1] You can also *store them elsewhere* if you like.

However, you'll notice that `--auto` needs a previous migration - our new app doesn't have one. Instead, in this case, we need to use `--initial`, which will create tables and indexes for all of the models in the app; it's what you use first, much like `syncdb`, and `--auto` is then used afterwards for each change.

So, let's create our first migration:

```
$ ./manage.py schemamigration southtut --initial
Creating migrations directory at '/home/andrew/Programs/litret/southtut/migrations'...
Creating __init__.py in '/home/andrew/Programs/litret/southtut/migrations'...
 + Added model southtut.Knight
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate southtut
```

(If this fails complaining that `south_migrationhistory` does not exist, you forgot to run syncdb *after you installed South*.)

As you can see, that's created a migrations directory for us, and made a new migration inside it. All we need to do now is apply our new migration:

```
$ ./manage.py migrate southtut
 Running migrations for southtut:
 - Migrating forwards to 0001_initial.
 > southtut:0001_initial
 - Loading initial data for southtut.
```

With that, South has created the new table for our model; check if you like, and try adding a few Knights using `./manage.py shell`.

Changing the model

So far, we've done nothing that `syncdb` couldn't accomplish; time to change that (or rather, our model). Let's add another field to our model:

```
from django.db import models

class Knight(models.Model):
    name = models.CharField(max_length=100)
    of_the_round_table = models.BooleanField()
    dances_whenever_able = models.BooleanField()
```

Now, if we weren't using migrations, making this new column appear on our `southtut_knight` table would be annoying at best. However, with South, we need only do two, quick steps: make a migration for the change, then apply it.

First, make the new migration, using the –auto feature:

```
$ ./manage.py schemamigration southtut --auto
 + Added field dances_whenever_able on southtut.Knight
Created 0002_auto__add_field_knight_dances_whenever_able.py. You can now apply this migration with:
```

(Notice that South has automatically picked a name for this migration; you can instead give migrations custom names by providing it as another argument)

Now, apply it:

```
$ ./manage.py migrate southtut
Running migrations for southtut:
 - Migrating forwards to 0002_auto__add_field_knight_dances_whenever_able.
 > southtut:0002_auto__add_field_knight_dances_whenever_able
 - Loading initial data for southtut.
```

With that, our new column is created; again, go and check, you'll be able to add Knights who can dance whenever they're able.

Converting existing apps

Sometimes, especially when introducting South into a project, you will want to use it for existing apps - ones for which the tables have already been created.

This is different from adding migrations to an all-new app, and you should see the *Converting An App* page for more information on how to do it.

Once you're happy with this basic usage of South, move on to *Part 2: Advanced Changes*.

2.4.2 Part 2: Advanced Changes

Now you've done a simple change to the model, let's look at some of the more advanced changes you can do with South.

Defaults

Firstly, let's deal with more tricky column types. In the previous part, we added a `BooleanField` to the table - this is easy for a database to handle, as it has a default value (of `False`) specified, so that's the value that gets used for the column in all of the existing rows.

However, some columns don't have a default defined. If the column is nullable - that is, `null=True` - then the existing rows will have NULL in the new column. Otherwise, if you've given no default, but the column is `NOT NULL` (i.e. `null=False`, the default), there's no value the database can put in the new column, and so you won't be able to reliably add the column [2].

If South detects such a situation, it will pop up and ask you what to do; let's make it do so.

First, change your model to add a new field that has no default, but is also not nullable:

```
from django.db import models

class Knight(models.Model):
    name = models.CharField(max_length=100)
    of_the_round_table = models.BooleanField()
    dances_whenever_able = models.BooleanField()
    shrubberies = models.IntegerField(null=False)
```

Now, let's try and get South to automatically generate a migration for that:

```
./manage.py schemamigration southtut --auto
 ? The field 'Knight.shrubberies' does not have a default specified, yet is NOT NULL.
 ? Since you are adding or removing this field, you MUST specify a default
 ? value to use for existing rows. Would you like to:
 ?  1. Quit now, and add a default to the field in models.py
 ?  2. Specify a one-off value to use for existing columns now
 ? Please select a choice:
```

South presents you with two options; if you select choice one, the command will quit without doing anything, and you should edit your `models.py` and add a default to the new field.

[2] Some database backends will let you add the column anyway if the table is empty, while some will refuse outright in this scenario.

If you select choice two, you'll get a Python prompt, where you should enter the default value you want to use for this migration. The default you enter will only ever be used for the currently-existing rows - this is a good option if you don't want the field on your model to have a default value.

We'll select choice two, and use `0` as our default (it is an IntegerField, after all):

```
 ? Please select a choice: 2
 ? Please enter Python code for your one-off default value.
 ? The datetime module is available, so you can do e.g. datetime.date.today()
 >>> 0
 + Added field shrubberies on southtut.Knight
Created 0003_auto__add_field_knight_shrubberies.py. You can now apply this migration with: ./manage.p
```

If you look at the generated migration, you'll see that there's a default specified for the new field, so your database won't cry. Finish off by running the migration:

```
$ ./manage.py migrate southtut
Running migrations for southtut:
 - Migrating forwards to 0003_auto__add_field_knight_shrubberies.
 > southtut:0003_auto__add_field_knight_shrubberies
 - Loading initial data for southtut.
```

Uniques

As well as detecting new fields (and also ones you've removed), South also detects most changes to fields, including changing their `unique` attributes.

First, let's make our Knights have unique names:

```
from django.db import models

class Knight(models.Model):
    name = models.CharField(max_length=100, unique=True)
    of_the_round_table = models.BooleanField()
    dances_whenever_able = models.BooleanField()
    shrubberies = models.IntegerField(null=False)
```

Run the automatic migration creator:

```
$ ./manage.py schemamigration --auto southtut
 + Added unique constraint for ['name'] on southtut.Knight
Created 0004_auto__add_unique_knight_name.py. You can now apply this migration with: ./manage.py migr
```

As you can see, it's detected the change in `unique`; you can now apply it:

```
$ ./manage.py migrate southtut
Running migrations for southtut:
 - Migrating forwards to 0004_auto__add_unique_knight_name.
 > southtut:0004_auto__add_unique_knight_name
 - Loading initial data for southtut.
```

South also detects changes to `unique_together` in your model's `Meta` in the same way.

ManyToMany fields

South should automatically detect ManyToMany fields; when you add the field, South will create the table the ManyToMany represents, and when you remove the field, the table will be deleted.

The one exception to this is when you have a 'through model' (i.e. you're using the `through=` option) - since the table for the model is already created when the model is detected, South does nothing with these types of ManyToMany fields.

Custom fields

If you've looked closely at the migration files, you'll see that South stores field definitions by storing their class, and the arguments that need to be passed to the field's constructor.

Since Python offers no way to get the arguments used in a class' constructor directly, South uses something called the *model introspector* to work out what arguments fields were passed. This knows what variables the arguments are stored into on the field, and using this knowledge, can reconstruct the arguments directly.

Because custom fields (either those written by you, or included with third-party apps) are all different, South can't work out how to get their arguments without extra help, so if you try to add, change or remove custom fields, South will bail out and say that you need to give it rules for your custom fields; this topic is covered in detail in *Custom Fields*.

More?

South supports most operations you'll do on your models day-to-day; if you're interested, there's a *full list of what the autodetector supports*.

You'll probably want to read *Part 3: Advanced Commands and Data Migrations* next.

2.4.3 Part 3: Advanced Commands and Data Migrations

Iteratively working on a migration

Sometimes, you'll find that you've made model changes that need to be further refined. Say you define this model:

```
class Group(models.Model):
    name = models.TextField(verbose_name="Name")
    facebook_page__id = models.CharField(max_length=255)
```

and you've created and applied this migration:

```
./manage.py schemamigration southtut --auto
./manage.py migrate southtut
```

You then notice two things: One, `name` should really be a `CharField`, not a `TextField`; and `facebook_page__id` contains double underscores where there should be a single one. You can fix these issues in your model, and then run:

```
./manage.py schemamigration southtut --auto --update
 + Added model southtut.Group
Migration to be updated, 0026_auto__add_group, is already applied, rolling it back now...
previous_migration: 0025_auto__foo (applied: 2012-05-25 21:20:47)
Running migrations for southtut:
  - Migrating backwards to just after 0025_auto__foo.
  < partner:0026_auto__add_group
Updated 0026_auto__add_group.py. You can now apply this migration with: ./manage.py migrate southtut
```

What happened here is that South removed the most recent migration, which created the model, but included the mistakes that were made, and replaced it with a new migration that includes the latest corrections made to the model.

It also noticed that the migration had already been applied, and automatically rolled it back for you. You can now apply the latest version of the migration to create the correct version of the model:

```
./manage.py migrate southtut
```

You may repeat this process as often as required to iron out any issues and come up with the final database changes required; which you can then publish, neatly packed into a single migration.

Listing current migrations

It can be very useful to know what migrations you currently have applied, and which ones are available. For this reason, there's `./manage.py migrate --list`.

Run against our project from before, we get:

```
$ ./manage.py migrate --list

southtut
 (*) 0001_initial
 (*) 0002_auto__add_field_knight_dances_whenever_able
 (*) 0003_auto__add_field_knight_shrubberies
 (*) 0004_auto__add_unique_knight_name
```

The output has an asterisk `(*)` next to a migration name if it has been applied, and an empty space `( )` if not [3].

If you have a lot of apps or migrations, you can also specify an app name to show just the migrations from that app.

Data migrations

The previous parts have only covered *schema migrations* - migrations which change the layout of your columns and indexes. There's also another kind of migration, the so-called *data migration*.

Data migrations are used to change the data stored in your database to match a new schema, or feature. For example, if you've been storing passwords in plain text [4], and you're moving to salted and hashed passwords, you might have these three steps (where each step corresponds to a migration):

- Create two new columns, `password_salt` and `password_hash` (a schema migration).
- Using the contents of the old `password` column, calculate salts and hashes for each user (a data migration)
- Remove the old `password` column (a schema migration).

The first and last migrations you already know how to do; make the relevant changes in the models.py file, and run `./manage.py schemamigration --auto myapp`. Remember that you need to add the two columns separately to deleting the old column, as otherwise the old column won't be around for us to get data out of, and you'll have lost all your users' passwords [5].

Let's follow a real example. Make a new app, and call it `southtut2`. Add it to `INSTALLED_APPS`, and then give it this model:

```
from django.db import models

class User(models.Model):

    username = models.CharField(max_length=255)
```

[3] An interesting side effect of this is that you can run the command `./manage.py migrate --list |grep -v "*"` to see which migrations are unapplied, and need running.
[4] If you're actually storing passwords in plaintext, please convert. Now.
[5] Always, always, backup your database before doing any kind of potentially destructive migration. One time, it *will* go wrong.

```
    password = models.CharField(max_length=60)
    name = models.TextField()
```

Make an initial migration for it, apply it, and then add a record:

```
$ ./manage.py schemamigration --initial southtut2
Creating migrations directory at '/home/andrew/Programs/litret/southtut2/migrations'...
Creating __init__.py in '/home/andrew/Programs/litret/southtut2/migrations'...
+ Added model southtut2.User
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate southtut2

$ ./manage.py migrate southtut2
Running migrations for southtut2:
 - Migrating forwards to 0001_initial.
 > southtut2:0001_initial
 - Loading initial data for southtut2.

$ ./manage.py shell
In [1]: from southtut2.models import User

In [2]: User.objects.create(username="andrew", password="ihopetheycantseethis", name="Andrew Godwin")
Out[2]: <User: User object>

In [3]: User.objects.get(id=1).password
Out[3]: u'ihopetheycantseethis'
```

As you can see, the password is clearly visible, which isn't good. Let's move to password hashing, while keeping everyone's password valid. Firstly, modify the model so it looks like this:

```
from django.db import models
import sha

class User(models.Model):

    username = models.CharField(max_length=255)
    password = models.CharField(max_length=60)
    password_salt = models.CharField(max_length=8, null=True)
    password_hash = models.CharField(max_length=40, null=True)
    name = models.TextField()

    def check_password(self, password):
        return sha.sha(self.password_salt + password).hexdigest() == self.password_hash
```

Make a schema migration that will create our two new columns (notice that they've both been added as `null=True`; once they have data, we'll alter them to be `null=False`):

```
$ ./manage.py schemamigration southtut2 --auto
 + Added field password_salt on southtut2.User
 + Added field password_hash on southtut2.User
Created 0002_auto__add_field_user_password_salt__add_field_user_password_hash.py. You can now apply t
```

Now, the second migration is more interesting. Firstly, we need to create a skeleton data migration (unlike schema migrations, South can't write these for you):

```
$ ./manage.py datamigration southtut2 hash_passwords
Created 0003_hash_passwords.py.
```

If you open up the file, you'll see that South has made the shell of a migration; the models definitions are there, the forwards() and backwards() functions are these, but there's no code in either. We'll write some code to port the passwords over in the forwards function:

```
def forwards(self, orm):
    import random, sha, string
    for user in orm.User.objects.all():
        user.password_salt = "".join([random.choice(string.letters) for i in range(8)])
        user.password_hash = sha.sha(user.password_salt + user.password).hexdigest()
        user.save()
```

Notice that we use `orm.User` to access the User model - this gives us the version of User from when this migration was created, so if we want to run the migration in future, it won't get a completely different, new, User model.

If you want to access models from other apps in your data migration, use a syntax like `orm['contenttypes.ContentType']`. Models will be available if you can somehow get to them via ForeignKey or ManyToMany traversal from your app's models; if you want to freeze other models, simply pass `--freeze appname` on the `datamigration` command line.

We should also raise an error in the `backwards()` method, since this process is by its very nature irreversible:

```
def backwards(self, orm):
    raise RuntimeError("Cannot reverse this migration.")
```

That looks good. Finally, remove the `password` field from your model, and run `schemamigration` one last time to make a migration to remove that field:

```
$ ./manage.py schemamigration southtut2 --auto
 ? The field 'User.password' does not have a default specified, yet is NOT NULL.
 ? Since you are adding or removing this field, you MUST specify a default
 ? value to use for existing rows. Would you like to:
 ?  1. Quit now, and add a default to the field in models.py
 ?  2. Specify a one-off value to use for existing columns now
 ? Please select a choice: 2
 ? Please enter Python code for your one-off default value.
 ? The datetime module is available, so you can do e.g. datetime.date.today()
 >>> ""
 - Deleted field password on southtut2.User
Created 0004_auto__del_field_user_password.py. You can now apply this migration with: ./manage.py mig
```

Notice that South is asking for a default value for `password`; if you were to reverse this migration, it tries to re-add the `password` column, and thus needs either a default value or for the field to be `null=True`. Here, I've fed it the empty string, as that's a reasonable default in this case.

Finally, let's apply all three migrations:

```
$ ./manage.py migrate southtut2
Running migrations for southtut2:
 - Migrating forwards to 0004_auto__del_field_user_password.
 > southtut2:0002_auto__add_field_user_password_salt__add_field_user_password_hash
 > southtut2:0003_hash_passwords
 > southtut2:0004_auto__del_field_user_password
 - Loading initial data for southtut2.
```

Looks good - we've added the new columns, migrated the passwords over, and then deleted the old column. Let's check our data was preserved:

```
$ ./manage.py shell
In [1]: from southtut2.models import User

In [2]: User.objects.get(id=1).check_password("ihopetheycantseethis")
Out[2]: True
```

```
In [3]: User.objects.get(id=1).check_password("fakepass")
Out[3]: False
```

That looks like a successful data migration!

You can do a lot more with this inside a data migration; any model can be available to you. The only caveat is that you won't have access to any custom methods or managers on your models, as they're not preserved as part of the freezing process (there's no way to do this generally); you'll have to copy any code you want into the migration itself. Feel free to make them methods on the `Migration` class; South ignores everything apart from `forwards` and `backwards`.

2.4.4 Part 4: Custom Fields

South 0.7 introduced a reasonably radical change from previous versions. Before, if you had a custom field, South would attempt to use magic [6] to determine how to freeze that field, so it could be recreated in a migration.

While this worked surprisingly well for most people, in a small percentage of cases it would get it completely wrong - even worse, you wouldn't know it was wrong until things changed a few weeks later. In the interests of both sanity and having less magic, you must now tell South how to freeze your custom fields.

Don't worry, it's pretty easy, and you only have to do it once per field.

Our Field

In this example, we'll be using a custom field which stores a list of tags in the database. We'll just store them in a TEXT column, with some delimiter separating the values (by default, we'll use `|`, but they can pass in something else as a keyword argument).

Here's the field class; in my code, I put this in `appname/fields.py` (for more on writing custom fields, see the Django docs):

```
from django.db import models

class TagField(models.TextField):

    description = "Stores tags in a single database column."

    __metaclass__ = models.SubfieldBase

    def __init__(self, delimiter="|", *args, **kwargs):
        self.delimiter = delimiter
        super(TagField, self).__init__(*args, **kwargs)

    def to_python(self, value):
        # If it's already a list, leave it
        if isinstance(value, list):
            return value

        # Otherwise, split by delimiter
        return value.split(self.delimiter)

    def get_prep_value(self, value):
        return self.delimiter.join(value)
```

To tell South about a custom field, you need to tell it two things; that this particular class is OK to use, and how to reconstruct the keyword arguments from a Field instance.

[6] And not very nice magic, either; a combination of regexes and the python `parser` module.

Keyword Arguments

South freezes fields by storing their class name and module (so it can get the field class itself) and the keyword arguments you used for that particular instance (for example, `CharField(max_length=50)` is a different database type to `CharField(max_length=150)`).

Since Python doesn't store the keyword arguments a class was passed, South has to reconstruct them using the field instance. For example, we know that `CharField`'s `max_length` attribute is stored as `self.max_length`, while `ForeignKeys` store their `to` attribute (the model they point to - also the first positional argument) as `self.rel.to`.

South knows all these rules for the core Django fields, but you need to tell it about your own ones. The good news is that South will trace the inheritance tree of your field class and add on rules from parent classes it knows about - thus, you only need tell South about extra keyword arguments you've added, not every possible argument the field could have.

In our example, we've only specified one extra keyword: `delimiter`. Here's the code we'd add for South to work with our new field; I'll explain it in a minute:

```
from south.modelsinspector import add_introspection_rules
add_introspection_rules([
    (
        [TagField], # Class(es) these apply to
        [],         # Positional arguments (not used)
        {           # Keyword argument
            "delimiter": ["delimiter", {"default": "|"}],
        },
    ),
], ["^southtut\.fields\.TagField"])
```

As you can see, to tell South about your new fields, you need to call the `south.modelsinspector.add_introspection_rules` function. You should put this code next to the definition of your field; the last thing you want is for the field to get imported, but for this code to not run.

`add_introspection_rules` takes two arguments; a list of rules, and a list of regular expressions. The regular expressions are used by South to see if a field is allowed to be introspected; just having a rule that matches it isn't enough, as rule inheritance means that any custom field class will have at least some rules on it (as they will inherit from `Field`, if not something more specific like `CharField`), and some custom fields can get by with only those inherited rules (more on that shortly).

The first argument is the list of rules. Each rule is a tuple (or list) with three items:

- A list of classes these rules apply to. You'll almost certainly have just `[MyField]` here.
- Positional argument specification. This should always be left blank, as an empty list - `[]`.
- Keyword argument specification. This is a dictionary, with the key being the name of the keyword argument, and the value being a tuple or list of `(attribute_name, options)`.

The attribute name says where the value of the keyword can be found - in our case, it's `'delimiter'`, as we stored our keyword in `self.delimiter`. (If this was the `ForeignKey` rule, we'd put `'rel.to'` here)

`options` is a dictionary. You can safely leave it blank, but to make things nicer, we can use it to specify the default value of this keyword - if the value South finds matches this, it will leave out this keyword from the frozen definition. This helps keep the frozen definitions shorter and more readable.

Simple Inheritance

If your field inherits directly from another Django field - say `CharField` - and doesn't add any new keyword arguments, there's no need to have any rules in your `add_introspection_rules`; you can just tell South that the field is alright as it is:

```
class UpperCaseField(models.TextField):
    "Makes sure its content is always upper-case."

    def to_python(self, value):
        return value.upper()

    def get_prep_value(self, value):
        return value.upper()

from south.modelsinspector import add_introspection_rules
add_introspection_rules([], ["^southtut\.fields\.UpperCaseField"])
```

More Information

There's more documentation on this subject, and on all the possible options, in the *Extending Introspection* section.

2.4.5 Part 5: Teams and Workflow

Migrations are all about improving the workflow for the developers and database administrators of projects, and we think it's very important that it doesn't add too much overhead to your daily coding, while at the same time reducing headaches caused by the inevitable changes in schema every project has.

Firstly, note that migrations aren't a magic bullet. If you've suddenly decided you're going to rearchitect your entire database schema, it might well be easier to not write migrations and just start again, especially if you have no production sites using the code (if you do, you might find custom serialisation/unserialisation to be a better way of saving your data).

With that in mind, migrations are really something you should be using the rest of the time. Hopefully, the previous parts of the tutorial have got you familiar with what can easily be achieved with them; we've tried to cover a good percentage of use cases, and if you think something should be included, don't hesitate to ask for it.

Developer Workflow

As a developer, you should be doing things in this order:

- Make the change to your models.py file (and affected code, such as post_syncdb signal hooks)
- Make the migration
- Rinse, repeat.

Don't try to make migrations before you make the changes; this will both invalidate the frozen model data on the migration and make startmigration –auto think nothing has changed. If you're making a large change, and want to split it over several migrations, do each schema change to models.py separately, then make the migration, and then make the next small change.

Note that the `./manage.py schemamigration` command has an `--update` mode that allows you to further iteratively refine your migration as such changes become necessary while working on your code. It is preferable to distribute a single migration for each atomic code change (a particular bug fixed, a new feature), then half a dozen migrations that could be merged into one. Remember that the purpose of migrations is to replay database changes on multiple machines; a separate migration is not required for changes that have only been applied locally.

Team Workflow

While migrations for an individual developer are useful, teams are perhaps the real reason they exist. It's very likely more than one member of your team will be making database changes, and migrations allow the other developers to apply their schema changes effortlessly and reproducibly.

You should keep all of your migrations in a VCS (for obvious reasons), and encourage developers to run `./manage.py migrate` if they see a new migration come in when they do an update or pull.

The issue with teams and migrations occurs when more than one person makes a migration in the same timeslot, and they both get committed without the other having been applied. This is analogous to two people editing the same file in a VCS at the same time, and like a VCS, South has ways of resolving the problem.

If this happens, the first thing to note is that South will detect the problem, and issue a message like this:

```
Inconsistent migration history
The following options are available:
    --merge: will just attempt the migration ignoring any potential dependency conflicts.
```

If you re-run migrate with `--merge`, South will simply apply the migrations that were missing out-of-order. This usually works, as teams are working on separate models; if it doesn't, you'll need to look at the actual migration changes and resolve them manually, as it's likely they'll conflict.

The second thing to note is that, when you pull in someone else's model changes complete with their own migration, you'll need to make a new empty migration that has the changes from both branches of development frozen in (if you've used mercurial, this is equivalent to a merge commit). To do so, simply run:

```
./manage.py schemamigration --empty appname merge_models
```

(Note that merge_models is just a migration name; change it for whatever you like)

The important message here is that *South is no substitute for team coordination* - in fact, most of the features are there purely to warn you that you haven't coordinated, and the simple merging on offer is only there for the easy cases. Make sure your team know who is working on what, so they don't write migrations that affect the same parts of the DB at the same time.

Complex Application Sets

It's often the case that, with Django projects, there is a set of apps which references each others' models.py files. This is, at its truest form, a dependency, and to ensure your migrations for such sets of applications apply sanely (i.e. the migrations that create the tables in one app happen before the migration that adds ForeignKeys to them in another app), South has a *Dependencies* feature. Once you've added dependencies to your migrations, South will ensure all prerequisites of a migration are applied before applying the migration itself.

2.5 Database API

South ships with a full database-agnostic API for performing schema changes on databases, much like Django's ORM provides data manipulation support.

Currently, South supports:

- PostgreSQL
- MySQL
- SQLite
- Microsoft SQL Server (beta support)

- Oracle (alpha support)

2.5.1 Accessing The API

South automatically exposes the correct set of database API operations as `south.db.db`; it detects which database backend you're using from your Django settings file. It's usually imported using:

```
from south.db import db
```

If you're using multiple database support (Django 1.2 and higher), there's a corresponding `south.db.dbs` dictionary which contains a DatabaseOperations object (the object which has the methods defined above) for each database alias in your configuration file:

```
from south.db import dbs
dbs['users'].create_table(...)
```

You can tell which backend you're talking to inside of a migration by examining `db.backend_name` - it will be one of `postgres`, `mysql`, `sqlite3`, `pyodbc` or `oracle`.

2.5.2 Database-Specific Issues

South provides a large amount of features, and not all features are supported by all database backends.

- PostgreSQL supports all of the South features; if you're unsure which database engine to pick, it's the one we recommend for migrating on.
- MySQL doesn't have transaction support for schema modification, meaning that if a migration fails to apply, the database is left in an inconsistent state, and you'll probably have to manually fix it. South will try and sanity-check migrations in a dry-run phase, and give you hints of what to do when it fails, however.
- SQLite doesn't natively support much schema altering at all, but South has workarounds to allow deletion/altering of columns. Unique indexes are still unsupported, however; South will silently ignore any such commands.
- SQL Server has been supported for a while, and works in theory, but the implementation itself may have bugs, as it's a contributed module and isn't under primary development. Patches and bug reports are welcome.
- Oracle is a new module as of the 0.7 release, and so is very much alpha. The most common operations work, but others may be missing completely; we welcome bug reports and patches against it (as with all other modules).

2.5.3 Methods

These are how you perform changes on the database. See *Accessing The API* to see how to get access to the `db` object.

db.add_column

```
db.add_column(table_name, field_name, field, keep_default=True)
```

Adds a column called `field_name` to the table `table_name`, of the type specified by the field instance field.

If `keep_default` is True, then any default value specified on the field will be added to the database schema for that column permanently. If not, then the default is only used when adding the column, and then dropped afterwards.

Note that the default value for fields given here is only ever used when adding the column to a non-empty table; the default used by the ORM in your application is the one specified on the field in your models.py file, as Django handles adding default values before the query hits the database.

The only case where having the default stored in the database as well would make a difference would be where you are interacting with the database from somewhere else, or Django doesn't know about the added column at all.

Also, note that the name you give for the column is the **field name**, not the column name - if the field you pass in is a ForeignKey, for example, the real column name will have _id on the end.

Examples

A normal column addition (the column is nullable, so all existing rows will have it set to NULL):

```
db.add_column('core_profile', 'height', models.IntegerField(null=True))
```

Providing a default value instead, so all current rows will get this value for 'height':

```
db.add_column('core_profile', 'height', models.IntegerField(default=-1))
```

Same as above, but the default is not left in the database schema:

```
db.add_column('core_profile', 'height', models.IntegerField(default=-1), keep_default=False)
```

db.alter_column

```
db.alter_column(table_name, column_name, field, explicit_name=True)
```

Alters the column `column_name` on the table `table_name` to match `field`. Note that this cannot alter all field attributes; for example, if you want to make a field `unique=True`, you should instead use `db.add_index` with `unique=True`, and if you want to make it a primary key, you should look into `db.drop_primary_key` and `db.create_primary_key`.

If explicit_name is false, ForeignKey? fields will have _id appended to the end of the given column name - this lets you address fields as they are represented in the model itself, rather than as the column name.

Examples

A simple change of the length of a VARCHAR column:

```
# Assume the table was created with name = models.CharField(max_length=50)
db.alter_column('core_nation', 'name', models.CharField(max_length=200))
```

We can also change it to a compatible field type:

```
db.alter_column('core_nation', 'name', models.TextField())
```

If we have a ForeignKey? named 'user', we can address it without the implicit '_id' on the end:

```
db.alter_column('core_profile', 'user', models.ForeignKey(orm['auth.User'], null=True, blank=True), e
```

Or you can specify the same operation with an explicit name:

```
db.alter_column('core_profile', 'user_id', models.ForeignKey(orm['auth.User'], null=True, blank=True)
```

db.clear_table

```
db.clear_table(table_name)
```

Deletes all rows from the table (truncation). Never used by South's autogenerators, but can prove useful if you're writing data migrations.

Examples

Clear all cached geocode results, as the schema is changing:

```
db.clear_table('core_geocoded')
db.add_column('core_geocoded', ...)
```

db.commit_transaction

```
db.commit_transaction()
```

Commits the transaction started at a `db.start_transaction` call.

db.create_index

```
db.create_index(table_name, column_names, unique=False, db_tablespace='')
```

Creates an index on the list of columns `column_names` on the table `table_name`.

By default, the index is simply for speed; if you would like a unique index, then specify `unique=True`, although you're better off using `db.create_unique` for that.

`db_tablespace` is an Oracle-specific option, and it's likely you won't need to use it.

Examples

Creating an index on the 'name' column:

```
db.create_index('core_profile', ['name'])
```

Creating a unique index on the combination of 'name' and 'age' columns:

```
db.create_index('core_profile', ['name', 'age'], unique=True)
```

db.create_primary_key

```
db.create_primary_key(table_name, columns)
```

Creates a primary key spanning the given `columns` for the table. Remember, you can only have one primary key per table; use `db.delete_primary_key` first if you already have one.

Examples

Swapping from the `id` to `uuid` as a primary key:

```
db.delete_primary_key('core_upload')
db.create_primary_key('core_upload', ['uuid'])
```

Adding a new composite primary key on "first name" and "last name":

```
db.create_primary_key('core_people', ['first_name', 'last_name'])
```

db.create_table

```
db.create_table(table_name, fields)
fields = ((field_name, models.SomeField(somearg=4)), ...)
```

This call creates a table called *table_name* in the database with the schema specified by *fields*, which is a tuple of `(field_name, field_instance)` tuples.

Note that this call will not automatically add an id column; you are responsible for doing that.

We recommend you create calls to this function using `schemamigration`, either in `--auto` mode, or by using `--add-model`.

Examples

A simple table, with one field, name, and the default id column:

```
db.create_table('core_planet', (
    ('id', models.AutoField(primary_key=True)),
    ('name', models.CharField(unique=True, max_length=50)),
))
```

A more complex table, which uses the ORM Freezer for its foreign keys:

```
db.create_table('core_nation', (
    ('name', models.CharField(max_length=255)),
    ('short_name', models.CharField(max_length=50)),
    ('slug', models.SlugField(unique=True)),
    ('planet', models.ForeignKey(orm.Planet, related_name="nations")),
    ('flag', models.ForeignKey(orm.Flag, related_name="nations")),
    ('planet_name', models.CharField(max_length=50)),
    ('id', models.AutoField(primary_key=True)),
))
```

db.create_unique

```
create_unique(table_name, columns)
```

Creates a unique index or constraint on the list of columns `columns` on the table `table_name`.

Examples

Declare the pair of fields `first_name` and `last_name` to be unique:

```
db.create_unique('core_people', ['first_name', 'last_name'])
```

db.delete_column

```
db.delete_column(table_name, column_name)
```

Deletes the column `column_name` from the table `table_name`.

Examples

Delete a column from a table:

```
db.delete_column('core_nation', 'title')
```

db.delete_index

```
db.delete_index(table_name, column_names, db_tablespace='')
```

Deletes an index created by db.create_index or one of the other South functions. Pass the column_names in exactly the same order as the other call to ensure this works; we use a hashing algorithm to make sure you can delete migrations by only specifying column names.

db_tablespace is an Oracle-specific option.

Examples

Deleting an index on 'name':

```
db.delete_index('core_profile', ['name'])
```

Deleting the unique index on the combination of 'name' and 'age' columns (from the db.create_index examples):

```
db.delete_index('core_profile', ['name', 'age'])
```

db.delete_foreign_key

```
delete_foreign_key(table_name, column)
```

Drops any foreign key constraints on the given column, if the database backend supported them in the first place.

Examples

Remove the foreign key constraint from user_id:

> db.delete_foreign_key('core_people', 'user_id')

db.delete_primary_key

```
db.delete_primary_key(table_name)
```

Deletes the current primary key constraint on the table. Does not remove the columns the primary key was using.

Examples

Swapping from the `id` to `uuid` as a primary key:

```
db.delete_primary_key('core_upload')
db.create_primary_key('core_upload', ['uuid'])
```

db.delete_table

```
db.delete_table(table_name, cascade=True)
```

Deletes (drops) the named table from the database. If cascade is True, drops any related constraints as well.

Examples

Usual call:

```
db.delete_table("core_planet")
```

Not cascading (beware, may fail):

```
db.delete_table("core_planet", cascade=False)
```

db.delete_unique

```
delete_unique(table_name, columns)
```

Deletes a unique index or constraint on the list of columns `columns` on the table `table_name`. The constraint/index. must already exist.

Examples

Declare the pair of fields `first_name` and `last_name` to no longer be unique:

```
db.delete_unique('core_people', ['first_name', 'last_name'])
```

db.execute

```
db.execute(sql, params=[])
```

Executes the **single** raw SQL statement `sql` on the database; optionally use params to replace the %s instances in sql (this is the recommended way of doing parameters, as it escapes them correctly for all databases).

If you want to execute a series of SQL statements instead, use `db.execute_many`.

Note that you should avoid using raw SQL wherever possible, as it will break the database abstraction in many cases. If you want to handle data, consider using the ORM Freezer, and remember that many operations such as creating indexes and changing primary keys have functions in the DB layer.

If there's a common operation you'd like to see added to the DB abstraction layer in South, consider asking on the mailing list or creating a ticket.

Examples

VACUUMing a table:

```
db.execute("VACUUM ANALYZE core_profile")
```

Updating values (this sort of task should really be done using the frozen ORM):

```
db.execute("UPDATE core_profile SET name = %s WHERE name = %s", ["andy", "andrew"])
```

db.execute_many

```
db.execute_many(sql, regex=r"(?mx) ([^';]* (?:'[^']*'[^';]*)*)", comment_regex=r"(?mx) (?:^\s*$)|(?:
```

Executes the given multi-statement SQL string `sql`. The two parameters are the regular expressions for splitting up statements (`regex`) and removing comments (`comment_regex`). We recommend you leave these at their default values, as they work on almost all SQL files.

If you only want to execute a single SQL statement, consider using `db.execute`, as it offers parameter escaping, and the regexes sometimes get the splitting wrong.

Examples

Run the PostGIS initialisation file:

```
db.execute_many(open("/path/to/lwpostgis.sql").read())
```

db.rename_column

```
db.rename_column(table_name, column_name, new_column_name)
```

Renames the column `column_name` in table `table_name` to `new_column_name`.

Examples

Simple rename:

```
db.rename_column('core_nation', 'name', 'title')
```

db.rename_table

```
db.rename_table(table_name, new_table_name)
```

Renames the table table_name to the new name new_table_name.

This won't affect what tables your models are looking for, of course; this is useful, for example, if you've renamed a model (and don't want to specify the old table name in Meta).

Examples

Simple rename:

```
db.rename_table('core_profile', 'core_userprofile')
```

db.rollback_transaction

```
db.rollback_transaction()
```

Rolls back the transaction started at a `db.start_transaction` call.

db.send_create_signal

```
db.send_create_signal(app_label, model_names)
```

Sends the post_syncdb signal for the given models `model_names` in the app `app_label`.

This signal is used by various bits of django internals - such as contenttypes - to hook new models into themselves, so you should really call it after the relevant `db.create_table` call. `startmigration` will add this automatically for you.

Note that the signals are not sent until the end of the whole migration sequence, so your handlers will not get called until all migrations are done. This is so that your handlers can deal with the most recent version of the model's schema, rather than the one in the migration where the signal is originally sent.

Examples

Sending a signal for the 'Profile' and 'Planet' models in my app 'core':

```
db.send_create_signal('core', ['Profile', 'Planet'])
```

db.start_transaction

```
db.start_transaction()
```

Wraps the following code (until it meets a `db.rollback_transaction` or `db.commit_transaction` call) in a transaction.

2.6 Converting An App

Converting an app to use South is very easy:

- Edit your settings.py and put 'south' into *INSTALLED_APPS* (assuming you've installed it to the right place)
- Run `./manage.py syncdb` to load the South table into the database. Note that syncdb looks different now - South modifies it.
- Run `./manage.py convert_to_south myapp` - South will automatically make and pretend to apply your first migration.

Note that you'll need to convert before you make any changes; South detects changes by comparing against the frozen state of the last migration, so it cannot detect changes from before you converted to using South.

2.6.1 Converting other installations and servers

The convert_to_south command only works entirely on the first machine you run it on. Once you've committed the initial migrations it made into your VCS, you'll have to run `./manage.py migrate myapp 0001 --fake` on every machine that has a copy of the codebase (make sure they were up-to-date with models and schema first).

(For the interested, this is required because the initial migration that convert_to_south makes will try and create all the existing tables; instead, you tell South that it's already applied using –fake, so the next migrations apply correctly.)

Remember that new installations of the codebase after this don't need these steps; you need only do a syncdb then a normal migrate.

2.7 Migration Structure

Migrations are, at the most basic level, files inside your app's migrations/ directory.

When South loads migrations, it loads all the python files inside migrations/ in ASCII sort order (e.g. 1 is before 10 is before 2), and expects to find a class called Migration inside each one, with at least a `forwards()` and `backwards()` method.

When South wants to apply a migration, it simply calls the `forwards()` method, and similarly when it wants to roll back a migration it calls `backwards()`. It's up to you what you do inside these methods; the usual thing is to do database changes, but you don't have to.

2.7.1 Sort Order

Since migrations are loaded in ASCII sort order, they won't be applied in the correct order if you call them `1_first, 2_second, ..., 10_tenth`. (10 sorts before 2).

Rather than force a specific naming convention, we suggest that if you want to use numerical migrations in this fashion (as we suggest you do) that you prefix the numbers with zeroes like so: `0001_first, 0002_second, 0010_tenth`.

All of South's automatic creation code will follow this scheme.

2.7.2 Transactions

Whenever `forwards()` or `backwards()` is called it is called inside a database transaction, which is committed if the method executes successfully or rolled back if it raises an error.

If you need to use two or more transactions inside a migration, either use two separate migrations (if you think it's appropriate), or have a snippet like this where you want a new transaction:

```
db.commit_transaction()      # Commit the first transaction
db.start_transaction()       # Start the second, committed on completion
```

Note that you must commit and start the next transaction if you are making both data and column changes. If you don't do this, you'll end up with your database hating you for asking it the impossible.

2.8 Dependencies

Migrations for apps are nice 'n all, but when you start writing a large project, with a lot of apps, you realise you have foreign key relationships between apps and working out what order migrations would need to be applied in for each app is just painful.

Luckily, we also had this problem, so South has a dependency system. Inside a migration, you can declare that it depends on having another app having run a certain migration first; for example, if my app "forum" depends on the "accounts" app having created its user profile table, we can do:

```
# forum/migrations/0002_post.py
class Migration:

    depends_on = (
        ("accounts", "0003_add_user_profile"),
    )

    def forwards(self):
        ....
```

Then, if you try and migrate to or beyond 0002_post in the forum app, it will first make sure accounts is migrated at least up to 0003_add_user_profile, and if not will migrate it for you.

Dependencies also work in reverse; South knows not to undo that 0003_add_user_profile migration until it has undone the 0002_post migration.

You can have multiple dependencies, and all sorts of wacky structures; there are, however, two rules:

- No circular dependencies (two or more migrations depending on each other)
- No upwards dependencies in the same app (so you can't make 0002_post in the forum app depend on 0003_room in the same app, either directly or through a dependency chain.

2.8.1 Reverse Dependencies

South also supports "reverse dependencies" - a dependecy where you say your migration must be run before another, rather than vice-versa. This is useful if you're trying to run a migration before another in a separate, third-party (or unchangeable) code.

Declaring these is just like the other kind, except you use needed_by:

```
class Migration:

    needed_by = (
        ("accounts", "0005_make_fks"),
    )

    def forwards(self):
        ....
```

2.9 Command Reference

South is mainly used via the console and its three important commands: migrate, schemamigration and datamigration. It also overrides a few parts of syncdb.

2.9.1 migrate

The migrate command is used to control the migration of the system forwards or backwards through the series of migrations for any given app.

The most common use is:

```
./manage.py migrate myapp
```

This will migrate the app myapp forwards through all the migrations. If you want to migrate all the apps at once, run:

```
./manage.py migrate
```

This has the same effect as calling the first example for every app, and will deal with Dependencies properly.

You can also specify a specific migration to migrate to:

```
./manage.py migrate myapp 0002_add_username
```

Note that, if the system has already migrated past the specified migration, it will roll back to it instead. If you want to migrate all the way back, specify the special migration name zero:

```
./manage.py migrate myapp zero
```

You can also just give prefixes of migrations, to save typing:

```
./manage.py migrate myapp 0002
```

But they must be unique:

```
$ ./manage.py migrate myapp 000
Running migrations for myapp:
 - Prefix 00 matches more than one migration:
     0001_initial
     0002_add_username
```

Options

- `--all`: Used instead of an app name, allows you to migrate all applications to the same target. For example, `./manage.py migrate --all --fake 0001` if you are converting a lot of apps.
- `--list`: Shows what migrations are available, and puts a * next to ones which have been applied.
- `--merge`: Runs any missed (out-of-order) migrations without rolling back to them.
- `--no-initial-data`: Doesn't load in any initial data fixtures after a full upwards migration, if there are any.
- `--fake`: Records the migration sequence as having been applied, but doesn't actually run it. Useful for *Converting An App*.
- `--db-dry-run`: Loads and runs the migration, but doesn't actually access the database (the SQL generated is thrown away at the last minute). The migration is also not recorded as being run; this is useful for sanity-testing migrations to check API calls are correct.

Conflict Resolution

South's migration system really comes into its own when you start getting conflicting migrations - that is, migrations that have been applied in the wrong sequence.

One example is if Anne writes new migrations 0003_foo and 0004_bar, runs the migration up to 0004 to make sure her local copy is up-to-date, and then updates her code from (say) Subversion. In the meantime, her coworker Bob has written a migration 0003_baz, which gets pulled in.

Now, there's a problem. 0003_baz should have been applied before 0004_bar, but it hasn't been; in this situation, South will helpfully say something like:

```
Running migrations for aeblog:
 - Current migration: 5 (after 0004_bar)
 - Target migration: 5 (after 0004_bar)
 ! These migrations should have been applied already, but aren't:
   - 0003_baz
 ! Please re-run migrate with one of these switches:
   --skip: Ignore this migration mismatch and keep going
   --merge: Just apply the missing migrations out of order
   If you want to roll back to the first of these migrations
   and then roll forward, do:
     ./manage.py migrate --skip 0002_add_username
     ./manage.py migrate
```

As you can see, you have two real options; `--merge`, which will just apply the missing migration and continue, and the two commands which roll back to before the missing migration (using `--skip` to ignore the error we're dealing with) and then migrating properly, in order, from there to the end.

Using `--skip` by itself will let you continue, but isn't much of a solution; South will still complain the next time you run a migrate without `--skip`.

Sometimes, even worse things happen and South finds out that an applied migration has gone missing from the filesystem. In this scenario, it will politely tell you to go fix the problem yourself, although in more recent versions, you also have the option to tell South to wipe all records of the missing migrations being applied.

Initial Data and post_syncdb

South will load initial_data files in the same way as syncdb, but it loads them at the end of every successful migration process, so ensure they are kept up-to-date, along with the rest of your fixtures (something to help ease the pain of migrating fixtures may appear shortly in South).

South also sends the post_syncdb signal when a model's table is first created (this functionality requires that you generated those migrations with startmigration). This behaviour is intended to mirror the behaviour of syncdb, although for sanity reasons you may want to consider moving any setup code connected to such a signal into a migration.

2.9.2 schemamigration

(In South 0.6 and below, this is called startmigration)

While migrate is the real meat and bones of South, schemamigration is by comparison an entirely optional extra. It's a utility to help write some of your migrations (specifically, the ones which change the schema) for you; if you like, you can ignore it and write everything youself, in which case we wish you good luck, and happy typing.

However, if you have a sense of reason, you'll realise that having the large majority of your migrations written for you is undoubtedly a good thing.

The main use of schemamigration is when you've just finished your shiny new models.py and want to load up your database. In vanilla Django, you'd just run syncdb - however, with migrations, you'll need a migration to create the tables.

In this scenario, you just run:

```
./manage.py schemamigration myapp --initial
```

That will write one big migration to create all the tables for the models in your app; just run `./manage.py migrate` to get it in and you're done in only one more step than syncdb!

Later on, you'll add models to your app, or change your fields. Each time you do this, run schemamigration with the –auto flag:

```
./manage.py schemamigration myapp --auto changed_user_model_bug_434
```

If you make further changes to your models, you can further refine the most recent migration:

```
./manage.py schemamigration myapp --auto --update
```

You can also manually specify changes:

```
./manage.py schemamigration mitest some_cols --add-field User.age --add-model User
```

See the tutorial for more.

Finally, if you're writing a schema migration that South can't automatically create for you (yet!) then you can just create a skeleton:

./manage.py schemamigration myapp my_new_column_migration –empty

Note that if you're writing a data migration, you should use the *datamigration* command instead.

Options

Note that you can combine as many `--add-X` options as you like.

- `--add-model`: Generates a creation migration for the given modelname.
- `--add-field`: Generates an add-column migration for modelname.field.
- `--add-index`: Generates an add-index migration for modelname.field.

- `--initial`: Like having –model for every model in your app. You should use this only for your first migration.
- `--auto`: Generates a migration with automatically-detected actions.
- `--update`: Update the most recent migration, instead of creating a new one.
- `--stdout`: Writes the migration to stdout instead of a file.

2.9.3 datamigration

(In South 0.6 and below, this is called startmigration)

When you want to create a data migration, use this command to create a blank template to write your migration with:

```
./manage.py datamigration books capitalise_titles
```

You can also freeze in additional apps if you want:

```
./manage.py datamigration books capitalise_titles --freeze awards
```

Options

- `--freeze`: Use appname to additional models into the app.
- `--stdout`: Writes the migration to stdout instead of a file.

2.9.4 graphmigrations

(New in South 0.7)

Run this command to generate a graphviz .dot file for your migrations; you can then use this to generate a graph of your migrations' dependencies.

Typical usage:

```
./manage.py graphmigrations | dot -Tpng -omigrations.png
```

This command can be particularly helpful to examine complex dependency sets between lots of different apps [7].

Options

This command has no options.

2.9.5 syncdb

South overrides the Django syncdb command; as well as changing the output to show apps delineated by their migration status, it also makes syncdb only work on a subset of the apps - those without migrations.

If you want to run syncdb on all of the apps, then use `--all`, but be warned; this will put your database schema and migrations out of sync. If you do this, you *might* be able to fix it with:

[7] This command was written and used for the first time while helping the debug the rather complex set of dependencies in django-cms; it's quite a sight to behold.

```
./manage.py migrate --fake
```

Options

- `--all`: Makes syncdb operate on all apps, not just unmigrated ones.

2.9.6 convert_to_south

An alias command that both creates an initial migration for an app and then fake-applies it. Takes one argument, the app label of the app to convert:

```
./manage.py convert_to_south myapp
```

There's more documentation on how to use this in the *Converting An App* section.

2.10 Unit Test Integration

By default, South's syncdb command will also apply migrations if it's run in non-interactive mode, which includes when you're running tests - it will run every migration every time you run your tests.

If you want the test runner to use syncdb instead of migrate - for example, if your migrations are taking way too long to apply - simply set `SOUTH_TESTS_MIGRATE = False` in settings.py.

2.10.1 South's own unit tests

South has its own set of unit tests; these will also be run when you run ./manage.py test. They do some fiddling with Django internals to set up a proper test environment; it's non-destructive, but if it's fouling up your own tests please submit a ticket about it.

You can also set `SKIP_SOUTH_TESTS=True` in settings.py to stop South's tests running, should they be causing issues.

2.11 ORM Freezing

South freezes the state of the ORM and models whenever you do a migration, meaning that when your migrations run in the future, they see the models and fields they're expecting (the ones that were around when they were created), rather than the current set (which could be months or even years newer).

This is accomplished by serialising the models into a large dictionary called `models` at the bottom of every migration. It's easy to see; it's the large chunk of dense code at the bottom.

2.11.1 Rationale behind the serialisation

South doesn't freeze every aspect of a model; for example, it doesn't preserve new managers, or custom model methods, as these would require serialising the python code that runs those method (and the code that depends on, and so forth).

If you want custom methods in your migration, you'll have to copy the code in, including any imports it relies on to work. Remember, however, for every import that you add, you're promising to keep that import valid for the life for the migration.

We also use a human-readable format that's easy to change; since South relies on the frozen models not only for reacreating the ORM but also for detecting changes, it's really useful to be able to edit them now and again (and also serves as a valuable debugging tool if you attach failing migrations to a ticket).

2.11.2 Serialisation format

`models` is a dict of `{'appname.modelname': fields}`, and `fields` is a dict of `{'fieldname': (fieldclass, positional_args, kwd_args)}`. `'Meta'` is also a valid entry in fields, in which case the value should be a dict of its attributes.

Make note that the entries in positional_args and kwd_args are **strings passed into eval**; thus, a string would be `'"hello"'`. We strongly recommend you use schemamigration/datamigration to freeze things.

2.11.3 Accessing the ORM

From inside a migration, you can access models from the frozen ORM in two ways. If the model you're accessing is part of the same app, you can simply call:

```
orm.ModelName
```

Otherwise, you'll need to specify the app name as well, using:

```
orm['myapp.ModelName']
```

For example, if you wanted to get a user with ID 1, you could use:

```
orm['auth.User'].objects.get(id=1)
```

Note that you can only access models that have been frozen; South automatically includes anything that could be reaches via foreign keys or many-to-many relationships, but if you want to add other models in, simply pass `--freeze appname` to the `./manage.py datamigration` command.

Also note that the `backwards()` method gets the ORM as frozen by the previous migration except for migrations that define `symmetrical = True` (new in South 1.0)

2.11.4 Frozen Meta Attributes

As well as freezing fields (for which South has a whole slew of rules on what to freeze - see *Extending Introspection*), it also freezes certain meta attributes of a model (the ones which we think will have an impact on the table schema or your frozen ORM use).

Currently, South freezes:

```
db_table
db_tablespace
unique_together
ordering
```

If there's something else you think should be frozen in the Meta, but which isn't, file a bug and we'll look into it.

2.12 Generic Relations

Generic relations' fields are be frozen, but unfortunately not the GenericForeignKey itself (see *ORM Freezing* for a reason why). To add it back onto a model, add the import for generic at the top of the migration and then in the body of forwards() put:

```
gfk = generic.GenericForeignKey()
gfk.contribute_to_class(orm.FooModel, "object_link")
```

This will add the GenericForeignKey onto the model as model.object_link. You can pass the optional content_type and id field names into the constructor as usual.

Also, be careful when using ContentType; make sure to use the frozen orm['contenttypes.ContentType'], and don't import it directly, otherwise comparisons may fail.

2.13 Custom Fields

2.13.1 The Problem

South stores field definitions by storing both their class and the arguments that need to be passed to the field's constructor, so it can recreate the field instance simply by calling the class with the stored arguments.

However, since Python offers no way to get the arguments used in a class' constructor directly, South uses something called the *model introspector* to work out what arguments fields were passed. This knows what variables the arguments are stored into on the field, and using this knowledge, can reconstruct the arguments directly.

This isn't the case for custom fields [8], however; South has never seen them before, and it can't guess at which variables mean what arguments, or what arguments are even needed; it only knows the rules for Django's internal fields and those of common third-party apps (those which are either South-aware, or which South ships with a rules module for, such as django-tagging).

2.13.2 The Solution

There are two ways to tell South how to work with a custom field; if it's similar in form to other fields (in that it has a set type and a few options) you'll probably want to *extend South's introspection rules*.

However, if it's particularly odd - such as a field which takes fields as arguments, or dynamically changes based on other factors - you'll probably find it easier to *add a south_field_triple method*.

2.14 Extending Introspection

(Note: This is also featured in the tutorial in *Part 4: Custom Fields*)

South does the majority of its field introspection using a set of simple rules; South works out what class a field is, and then runs all rules which have been defined for either that class or a parent class of it.

This way, all of the common options (such as `null=`) are defined against the main `Field` class (which all fields inherit from), while specific options (such as `max_length`) are defined on the specific fields they apply to (in this case, `CharField`).

[8] 'Custom Fields' in this context refers to any field that is not part of Django's core. GeoDjango fields are part of the core, but ones in third-party apps are 'custom'. Note also that a field is considered custom even if it inherits directly from a core field and doesn't override anything; there's no way for South to reliably tell that it does so.

If your custom field inherits from a core Django field, or another field for which there are already introspection rules, and it doesn't add any new attributes, then you probably won't have to add any rules for it, as it will inherit all those from its parents. In this case, a call like this should work:

```
from south.modelsinspector import add_introspection_rules
add_introspection_rules([], ["^myapp\.stuff\.fields\.SomeNewField"])
```

Note that you must always specify a field as allowed, even if specifies no new rules of its own - the alternative is that South must presume all fields without any new rules specified only have the options of their parents, which is wrong some of the time.

Thus, there are two stages to adding support for your custom field to South; firstly, adding some rules for the new arguments it introduces (or possibly not adding any), and secondly, adding its field name to the list of patterns South knows are safe to introspect.

2.14.1 Rules

Rules are what make up the core logic of the introspector; you'll need to pass South a (possibly empty) list of them. They consist of a tuple, containing:

- A tuple or list of one or more classes to which the rules apply (remember, the rules apply to the specified classes and all subclasses of them).
- Rules for recovering positional arguments, in order of the arguments (you are strongly advised not to use this feature, and use keyword argument instead).
- A dictionary of keyword argument rules, with the key being the name of the keyword argument, and the value being the rule.

Each rule is itself a list or tuple with two elements:

- The first element is the name of the attribute the value is taken from - if a field stored its max_length argument as `self.max_length`, say, this would be `"max_length"`.
- The second element is a (possibly empty) dictionary of options describing the various different variations on handling of the value.

An example (this is the South rule for the many-to-one relationships in core Django):

```
rules = [
  (
    (models.ForeignKey, models.OneToOneField),
    [],
    {
        "to": ["rel.to", {}],
        "to_field": ["rel.field_name", {"default_attr": "rel.to._meta.pk.name"}],
        "related_name": ["rel.related_name", {"default": None}],
        "db_index": ["db_index", {"default": True}],
    },
  )
]
```

You'll notice that you're allowed to have dots in the attribute name; ForeignKeys, for example, store their destination model as `self.rel.to`, so the attribute name is `"rel.to"`.

The various options are detailed below; most of them allow you to specify the default value for a parameter, so arguments can be omitted for clarity where they're not necessary. The one special case is the `is_value` keyword; if this is present and True, then the first item in the list will be interpreted as the actual value, rather than the attribute path to it on the field. For example:

```
"frozen_by_south": [True, {"is_value": True}],
```

Parameters

- default: The default value of this field (directly as a Python object). If the value retrieved ends up being this, the keyword will be omitted from the frozen result. For example, the base Field class' "null" attribute has {'default':False}, so it's usually omitted, much like in the models.
- default_attr: Similar to default, but the value given is another attribute to compare to for the default. This is used in to_field above, as this attribute's default value is the other model's pk name.
- default_attr_concat: For when your default value is even more complex, default_attr_concat is a list where the first element is a format string, and the rest is a list of attribute names whose values should be formatted into the string.
- ignore_if: Specifies an attribute that, if it coerces to true, causes this keyword to be omitted. Useful for `db_index`, which has `{'ignore_if': 'primary_key'}`, since it's always True in that case.
- ignore_dynamics: If this is True, any value that is "dynamic" - such as model instances - will cause the field to be omitted instead. Used internally for the `default` keyword.
- is_value: If present, the 'attribute name' is instead used directly as the value. See *above* for more info.

2.14.2 Field name patterns

The second of the two steps is to tell South that your field is now safe to introspect (as you've made sure you've added all the rules it needs).

Internally, South just has a long list of regular expressions it checks fields' classes against; all you need to do is provide extra arguments to this list.

Example (this is in the GeoDjango module South ships with, and presumes `rules` is the rules triple you defined previously):

```
from south.modelsinspector import add_introspection_rules
add_introspection_rules(rules, ["^django\.contrib\.gis"])
```

Additionally, you can ignore some fields completely if you know they're not needed. For example, django-taggit has a manager that actually shows up as a fake field (this makes the API for using it much nicer, but confuses South to no end). The django-taggit module we ship with contains this rule to ignore it:

```
from south.modelsinspector import add_ignored_fields
add_ignored_fields(["^taggit\.managers"])
```

2.14.3 Where to put the code

You need to put the call to `add_introspection_rules` somewhere where it will get called before South runs; it's probably a good choice to have it either in your `models.py` file or the module the custom fields are defined in.

2.14.4 General Caveats

If you have a custom field which adds other fields to the model dynamically (i.e. it overrides contribute_to_class and adds more fields onto the model), you'll need to write your introspection rules appropriately, to make South ignore the extra fields at migration-freezing time, or to add a flag to your field which tells it not to make the new fields again. An example can be found here.

2.15 south_field_triple

There are some cases where introspection of fields just isn't enough; for example, field classes which dynamically change their database column type based on options, or other odd things.

Note: *Extending the introspector* is often far cleaner and easier than this method.

The method to implement for these fields is `south_field_triple()`.

It should return the standard triple of:

```
('full.path.to.SomeFieldClass', ['positionalArg1', '"positionalArg2"'], {'kwarg':'"value"'})
```

(this is the same format used by the *ORM Freezer*; South will just use your output verbatim).

Note that the strings are ones that will be passed into eval, so for this reason, a variable reference would be `'foo'` while a string would be `'"foo"'`.

2.15.1 Example

Here's an example of this method for django-modeltranslation's TranslationField. This custom field stores the type it's wrapping in an attribute of itself, so we'll just use that:

```
def south_field_triple(self):
    "Returns a suitable description of this field for South."
    # We'll just introspect the _actual_ field.
    from south.modelsinspector import introspector
    field_class = self.translated_field.__class__.__module__ + "." + self.translated_field.__class__
    args, kwargs = introspector(self.translated_field)
    # That's our definition!
    return (field_class, args, kwargs)
```

2.16 The Autodetector

The autodetector is the part of South you'll probably be using the most, as well as being the feature that people seem to like the most.

The general use of the autodetector is covered in *Part 1: The Basics*; this is more of a reference of what it's capable of.

When the autodetector runs, it compares your current models with those frozen in your most recent migration on the app, and if it finds any changes, yields one or more Actions to the South migration-file-writer.

2.16.1 Supported Actions

Model creation and deletion

South will happily detect the creation and deletion of models; this is the oldest and most well-worn feature of the autodetector, and so has very few caveats.

One thing to note is that, while South calls the post_syncdb hook on your models (much like `syncdb` does), it calls it when it initially creates the table, not at the end of the migration, so your hook might well get called when the model doesn't have its full table.

Consider moving your hook code into its own data migration, or use one of our own *Signals*.

Field addition and removal

South detects addition and removal of fields fine, and should correctly create indexes and constraints for new fields.

Note that when you add or remove a field, you need a default specified; there's more explanation on this in the *Defaults* part of the tutorial.

Field changes

South will detect if you change a field, and should correctly change the field type, with one exception:

- If you alter to a field with a CHECK constraint (e.g. `PositiveIntegerField`) the constraint won't be added to the column (it is removed if you alter away, however). This will be fixed in a future release.

ManyToMany addition and removal

ManyToMany fields are detected on addition and removal; when you add the field, South will create the table the ManyToMany represents, and when you remove the field, the table will be deleted.

The one exception to this is when you have a 'through model' (i.e. you're using the `through=` option) - since the table for the model is already created when the model is detected, South does nothing with these types of ManyToMany fields.

Unique changes

If you change the `unique=` attribute on a field, or the `unique_together` in a model's Meta, South will detect and change the constraints on the database accordingly (except on SQLite, where we don't get have the code to edit UNIQUE constraints).

2.17 Signals

South offers its own signals, if you want to write code which executes before or after migrations. They're available from `south.signals`.

2.17.1 pre_migrate

Sent just before South starts running migrations for an app.

Provides one argument, `app`, a string containing the app's label.

2.17.2 post_migrate

Sent just after South successfully finishes running migrations for an app. Note that if the migrations fail in the middle of executing, this will not get called.

Provides one argument, `app`, a string containing the app's label.

2.17.3 ran_migration

Sent just after South successfully runs a single migration file; can easily be sent multiple times in one run of South, possibly hundreds of times if you have hundreds of migrations, and are doing a fresh install.

Provides three arguments, `app`, a string containing the app's label, `migration`, a string containing the name of the migration file without the file extension, and `method`, which is either `"forwards"` or `"backwards"`.

2.18 Fixtures

A few things change when you're using fixtures with South.

2.18.1 initial_data

Much like syncdb, South will load the initial_data fixture when an app has been successfully migrated to the latest migration for an app. Note that the data in the fixture will not be available before then; South only applies it at the end, as it may not match the current database schema.

2.18.2 Fixtures from migrations

If you need to load a fixture as part of your database setup - say, you have a migration that depends on it being around - the best thing to do is to write a new migration to load the fixture in. That way, the fixture will always be loaded at the correct time.

To make such a migration, first make a blank migration:

```
./manage.py datamigration appname load_myfixture
```

Then, open the new migration file, and restructure your forwards() method so it looks like this:

```
def forwards(self, orm):
    from django.core.management import call_command
    call_command("loaddata", "my_fixture.json")
```

(you'll have to leave backwards() empty, as there's not much you can do to reverse this).

Then, when this migration is run, it will load the given fixture.

2.19 Settings

South has its own clutch of custom settings you can use to tweak its operation. As with normal Django settings, these go in `settings.py`, or a variant thereof.

2.19.1 SKIP_SOUTH_TESTS

South has a somewhat fragile test suite, as it has to fiddle with `INSTALLED_APPS` at runtime to load in its own testing apps. If the South tests are failing for you, and you'd rather they be ignored (by your CI system or similar, in particlar) set this to `True`. Defaults to `False`.

2.19.2 SOUTH_DATABASE_ADAPTER

(Django 1.1 and below)

If set, overrides the database module South uses for generating DDL commands. Defaults to `south.db.<DATABASE_ENGINE>`.

2.19.3 SOUTH_DATABASE_ADAPTERS

(Django 1.2 and above)

A dictionary with database aliases as keys and the database module South will use as values. South defaults to using the internal `south.db.<ENGINE> modules`.

2.19.4 MySQL STORAGE_ENGINE

If (database-specific) `STORAGE_ENGINE` is set, South will tell MySQL to use the given storage engine for new items.

For Django version before 1.2 the (global) setting is `DATABASE_STORAGE_ENGINE`.

Example for Django 1.2 and above:

```
DATABASES = {
   'default': {
        ...
        'STORAGE_ENGINE': 'INNODB'
   }
}
```

For Django before 1.2:

```
DATABASE_STORAGE_ENGINE = 'INNODB'
```

2.19.5 SOUTH_AUTO_FREEZE_APP

When set, South freezes a migration's app and appends it to the bottom of the migration file (the default behaviour, and required for `--auto` to work). If you want to manually pass in `--freeze appname` instead, or just don't like the clutter, set this to `False`. Defaults to `True`.

2.19.6 SOUTH_TESTS_MIGRATE

If this is `False`, South's test runner integration will make the test database be created using syncdb, rather than via migrations (the default). Set this to `False` if you have migrations which take too long to migrate every time tests run, but be wary if you rely on migrations to do special things. Defaults to `True` in 0.7 and above, `False` in 0.6 and below.

2.19.7 SOUTH_LOGGING_ON

If this is True the SQL run by South is logged to a file. You must also set `SOUTH_LOGGING_FILE` to a valid file that you want to log to.

2.19.8 SOUTH_LOGGING_FILE

See SOUTH_LOGGING_ON for more info.

A sample setting would be:

```
SOUTH_LOGGING_FILE = os.path.join(os.path.dirname(__file__),"south.log")
```

2.19.9 SOUTH_MIGRATION_MODULES

(South 0.7 and higher)

A dictionary of alternative migration modules for apps. By default, apps look for their migrations in "<app-name>.migrations", but you can override this here, if you have project-specific migrations sets.

Note that the keys in this dictionary are 'app labels', not the full paths to apps; for example, were I to provide a migrations directory for `django.contrib.auth`, I'd want to use `auth` as the key here.

Example:

```
SOUTH_MIGRATION_MODULES = {
    'books': 'myproject.migrations.books',
}
```

Additionally, you can use this setting to turn off migrations for certain apps, by saying their migrations are in some nonexistent module; for example:

```
SOUTH_MIGRATION_MODULES = {
    'books': 'ignore',
}
```

2.19.10 SOUTH_USE_PYC

If set to `True`, South will also use .pyc files for migrations. Useful if you distribute your code only in .pyc format.

2.20 Release Notes

Release notes from various versions of South.

2.20.1 South 0.7

This is a major new release of South. A lot of work has been done to the internals, and a few annoying remnants from South's history have finally been eradicated.

Backwards incompatible changes

Tests now run with migrations by default, not using syncdb for everything as in 0.6. This is the behaviour most people expect; to turn it off again, set `SOUTH_TESTS_MIGRATE` to False (migrating everything can be slow).

In addition, you may note that some or all of your custom fields don't work when you upgrade; read more about this at *Custom Fields*. You may also wish to change your old migration files and insert the full path to custom field classes in the `models` dictionary entries, to prevent future issues.

Finally, migration names must now not contain any dashes (or other characters invalid in Python module names) - if they do, you'll need to rename them and also fix the appropriate entries in your south_migrationhistory table.

Major changes

Core Refactoring

The entire migration and dependency engine has been refactored to be class-based, rather than the mess of functions and variables it was before, and will now be a lot easier to maintain, as well as being nice and quick.

Much thanks to Simon Law for doing a lot of the legwork on this one.

Command Changes

The `startmigration` command (which used to be one massive file) has been removed, and refactored into new commands:

- `schemamigration`, which is very similar to the old `startmigration`
- `datamigration`, which should be used to create new data migrations

In addition, the `--model` argument to `startmigration` is now `--add-model` on `schemamigration`, for consistency with the other arguments, and `schemamigration` no longer requires a migration name; if you don't provide one, it will autogenerate a reasonably sensible one.

Finally, South now detects when you're adding a column that needs a default value, and prompts you for it, rather than crashing when you tried to apply the migration, like before.

Django Support

This version of South fully supports Django 1.2 (as well as 1.1 and 1.0), and has some limited multi-db functionality (migrate has gained a –database option) [9].

Custom Fields

Custom fields are no longer parsed if they don't introspect; instead, an error is raised every time. This is because parsing was causing scenarios where migrations sometimes worked, and then failed mysteriously later; the new solution means they'll always work or fail.

This does have the unfortunate side-effect of making South not "magically" make your simpler custom fields work any more; we're trying to help by shipping introspection modules for the more common third-party apps with South, but you may also want to read the new *reference for your own introspection rules*, or *our new tutorial chapter on it*.

Migration Directories

You can now set custom migration directories (actually done as Python modules) if you need per-project migrations for an app, or if you are using third-party apps and don't want to store the migrations with the app.

You simply need to set the new *SOUTH_MIGRATION_MODULES* setting.

[9] Note that multi-db functionality is unavailable if using South 0.7 with earlier versions of Django.

Supported Databases

SQLite now has full, near-bulletproof support for altering columns, deleting columns, and other basic operations SQLite doesn't support natively.

Oracle now has alpha support.

Migrations Files

Migrations files no longer import from appname.models; model classes are now referred to by their full path, and retrieved using `Migration.gf` - this means a field now looks like:

```
self.gf('django.db.models.fields.TextField')(blank=True)
```

Also, migration classes should now inherit from `south.v2.SchemaMigration` or `south.v2.DataMigration`. This doesn't do much at the moment, but is designed so we can easily change the migration API in future and keep backwards compatability.

Bugfixes and minor changes

There's also an assorted array of bugfixes; see the milestone status page for details.

Thanks

This release wouldn't have been possible without:

- Simon Law, who wrote most of the migration refactor and now knows too much about how our dependencies work
- Torchbox, who sponsored Andrew's work on the startmigration refactor, the rest of the migration refactor, and a lot of other small things.
- Ilya Roitburg, who contributed the Oracle database module.

2.20.2 South 0.7.1

This is a minor new release of South, and the first bugfix release for the *0.7 series*.

Backwards incompatible changes

None.

Changes

South tests

South's internal test suite now doesn't run by default (the `SKIP_SOUTH_TESTS` setting now defaults to True). This is mainly because the test suite is meant to be run in isolation (the test framework continually changes `INSTALLED_APPS` and fiddles with the ORM as it runs, among other things), and was causing compatability problems with other applications.

If you wish to run the tests still, simply set `SKIP_SOUTH_TESTS = False`.

Data Migrations

There was an annoying issue that caused failing data migrations under MySQL to suddenly run their backwards() method and produce an error completely unrelated to the original problem. This has been fixed.

Commands

`./manage.py migrate` has gained a new `--ignore-ghost-migrations`, which will temporarily silence South's complaining about missing migrations on disk if you really know what you're doing (i.e. temporary branch switching).

In addition, –noinput is now correctly respected for the "./manage.py migrate" command.

Dependencies

A bug and some nondeterminism in the new dependency engine has been fixed (previously, dependencies were sometimes calculated wrongly, and the non-determinism meant that this only happened on certain architectures).

Other changes

A whole assortment of minor bugs has been fixed; for the complete list, see the milestone in our Trac.

2.20.3 South 0.7.2

This is a minor new release of South, and the second bugfix release for the *0.7 series*.

Backwards incompatible changes

None.

Changes

Ordering of actions

A few issues with ordering of index deletion versus field/table deletion have now been fixed, so hopefully things will delete or migrate backwards first time.

blank

If you have a CharField or TextField with blank=True, you now no longer need to specify a default value. In addition, changes to blank no longer trigger an alteration migration for that field, since it doesn't affect the database.

Schemas

South should now work if you aren't using the 'public' schema; you'll need to set the SCHEMA database setting first, though.

Arguments

A bit of tidying up has been done for arguments; *migrate* now accepts *–noinput* and *convert_to_south* accepts the ghost migration options.

Other changes

A whole assortment of minor bugs has been fixed; for the complete list, see the milestone in our Trac.

2.20.4 South 0.7.3

This is a minor new release of South, and the third bugfix release for the *0.7 series*.

Backwards incompatible changes

None.

More NULL safety checks

South now also checks if you're converting a field to/from NULL and makes you add defaults as appropriate.

Circular Dependency Fixes

South's circular-dependency-checking engine has had some fixes to stop false positives.

PyODBC backend improvements

Thanks to Shai Berger, the MSSQL backend has had some much-needed improvements.

Various other improvements

Fixes to generated migration names, table name escaping, WSGI compatability, MySQL foreign key checks, and 2.4 compatability.

2.20.5 South 0.7.4

This is a minor new release of South, and the fourth bugfix release for the *0.7 series*. The main feature is compatability with Django 1.4.

It has unfortunately been over a year since the last South release - I intend to make them more frequent from now on, especially where large bugs are concerned. South 1.0 will eventually happen, but we may end up dropping support for some of the older Django versions when it's released.

Backwards incompatible changes

None.

Timezone support

Thanks to work by various contributors, most notably Jannis Leidel, South now supports timezone-aware datetime fields (as is the default in Django 1.4).

unique handling

South now correctly handles the use of the unique= keyword on columns.

on_delete handling

South now correctly handles and persists the on_delete argument to ForeignKeys.

Constraint caching

Runtime has been improved thanks to caching of constraints at runtime. Thanks to Jack Diederich for a large portion of this work.

Oracle and MSSQL fixes

Thanks mostly to the work of Shai Berger, the Oracle and MSSQl backends have been signficantly upgraded and improved.

Other fixes

As usual, there are tens of other minor fixes throughout the codebase. Full details of those are available on Trac.

2.20.6 South 0.7.5

This is a minor new release of South, and the fifth bugfix release for the *0.7 series*.

Compatability Notes

From now on, South will only officially be compatable with Django 1.2 and up, and thus only with Python 2.4 and up. Future versions of South are likely to require newer and newer versions of Django in order to simplify the codebase somewhat.

Backwards incompatible changes

None.

Index naming

Single-column indexes should now be named the same as if they were created using `syncdb`

UUIDs

UUID default values now work correctly on the PostgreSQL backend.

Transactions

Transactions now use the correct database in a multi-db setup - previously, they were sometimes using the default database instead of another configured one.

Recursive Foreign Keys

Deletion of self-referencing ForeignKeys is now possible again.

Unmanaged models

A bug with ignoring changes to unmanaged models has been fixed, so they are now ignored properly.

Oracle/SQL Server

A few minor fixes, including the ability to change TextFields to CharFields and vice-versa on the Oracle backend.

Other fixes

As usual, there are tens of other minor fixes throughout the codebase. Full details of those are available on Trac.