Django Documentation

Release 1.5

Django Software Foundation

October 30, 2013

CONTENTS

CHAPTER

ONE

DJANGO DOCUMENTATION

Everything you need to know about Django (and then some).

1.1 Getting help

Having trouble? We'd like to help!

- Try the *FAQ* – it's got answers to many common questions.
- Looking for specific information? Try the *genindex*, *modindex* or the *detailed table of contents*.
- Search for information in the archives of the django-users mailing list, or post a question.
- Ask a question in the #django IRC channel, or search the IRC logs to see if it's been asked before.
- Report bugs with Django in our ticket tracker.

1.2 First steps

- **From scratch:** *Overview* | *Installation*
- **Tutorial:** *Part 1* | *Part 2* | *Part 3* | *Part 4*

1.3 The model layer

- **Models:** *Model syntax* | *Field types* | *Meta options*
- **QuerySets:** *Executing queries* | *QuerySet method reference*
- **Model instances:** *Instance methods* | *Accessing related objects*
- **Advanced:** *Managers* | *Raw SQL* | *Transactions* | *Aggregation* | *Custom fields* | *Multiple databases*
- **Other:** *Supported databases* | *Legacy databases* | *Providing initial data* | *Optimize database access*

1.4 The template layer

- **For designers:** *Syntax overview* | *Built-in tags and filters*
- **For programmers:** *Template API* | *Custom tags and filters*

1.5 The view layer

- **The basics:** *URLconfs* | *View functions* | *Shortcuts* | *Decorators*
- **Reference:** *Request/response objects* | *TemplateResponse objects*
- **File uploads:** *Overview* | *File objects* | *Storage API* | *Managing files* | *Custom storage*
- **Class-based views:** *Overview* | *Built-in class-based views* | *Built-in view mixins*
- **Advanced:** *Generating CSV* | *Generating PDF*
- **Middleware:** *Overview* | *Built-in middleware classes*

1.6 Forms

- **The basics:** *Overview* | *Form API* | *Built-in fields* | *Built-in widgets*
- **Advanced:** *Forms for models* | *Integrating media* | *Formsets* | *Customizing validation*
- **Extras:** *Form preview* | *Form wizard*

1.7 The development process

- **Settings:** *Overview* | *Full list of settings*
- **Exceptions:** *Overview*
- **django-admin.py and manage.py:** *Overview* | *Adding custom commands*
- **Testing:** *Overview*
- **Deployment:** *Overview* | *WSGI servers* | *FastCGI/SCGI/AJP* | *Apache authentication* | *Handling static files* | *Tracking code errors by email*

1.8 Other batteries included

- *Admin site* | *Admin actions* | *Admin documentation generator*
- *Authentication*
- *Cache system*
- *Clickjacking protection*
- *Comments* | *Moderation* | *Custom comments*
- *Conditional content processing*
- *Content types and generic relations*
- *Cross Site Request Forgery protection*
- *Cryptographic signing*
- *Databrowse*
- *E-mail (sending)*

- *Flatpages*
- *GeoDjango*
- *Humanize*
- *Internationalization*
- *Jython support*
- *"Local flavor"*
- *Logging*
- *Messages*
- *Pagination*
- *Python 3 compatibility*
- *Redirects*
- *Security*
- *Serialization*
- *Sessions*
- *Signals*
- *Sitemaps*
- *Sites*
- *Static Files*
- *Syndication feeds (RSS/Atom)*
- *Unicode in Django*
- *Web design helpers*
- *Validators*

1.9 The Django open-source project

- **Community:** *How to get involved* | *The release process* | *Team of committers* | *The Django source code repository*
- **Design philosophies:** *Overview*
- **Documentation:** *About this documentation*
- **Third-party distributions:** *Overview*
- **Django over time:** *API stability* | *Release notes and upgrading instructions* | *Deprecation Timeline*

CHAPTER

TWO

GETTING STARTED

New to Django? Or to Web development in general? Well, you came to the right place: read this material to quickly get up and running.

2.1 Django at a glance

Because Django was developed in a fast-paced newsroom environment, it was designed to make common Web-development tasks fast and easy. Here's an informal overview of how to write a database-driven Web app with Django.

The goal of this document is to give you enough technical specifics to understand how Django works, but this isn't intended to be a tutorial or reference – but we've got both! When you're ready to start a project, you can *start with the tutorial* or *dive right into more detailed documentation*.

2.1.1 Design your model

Although you can use Django without a database, it comes with an object-relational mapper in which you describe your database layout in Python code.

The *data-model syntax* offers many rich ways of representing your models – so far, it's been solving two years' worth of database-schema problems. Here's a quick example, which might be saved in the file `mysite/news/models.py`:

```
class Reporter(models.Model):
    full_name = models.CharField(max_length=70)

    def __unicode__(self):
        return self.full_name

class Article(models.Model):
    pub_date = models.DateTimeField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)

    def __unicode__(self):
        return self.headline
```

2.1.2 Install it

Next, run the Django command-line utility to create the database tables automatically:

```
manage.py syncdb
```

The `syncdb` command looks at all your available models and creates tables in your database for whichever tables don't already exist.

2.1.3 Enjoy the free API

With that, you've got a free, and rich, *Python API* to access your data. The API is created on the fly, no code generation necessary:

```
# Import the models we created from our "news" app
>>> from news.models import Reporter, Article

# No reporters are in the system yet.
>>> Reporter.objects.all()
[]

# Create a new Reporter.
>>> r = Reporter(full_name='John Smith')

# Save the object into the database. You have to call save() explicitly.
>>> r.save()

# Now it has an ID.
>>> r.id
1

# Now the new reporter is in the database.
>>> Reporter.objects.all()
[<Reporter: John Smith>]

# Fields are represented as attributes on the Python object.
>>> r.full_name
'John Smith'

# Django provides a rich database lookup API.
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Reporter matching query does not exist. Lookup parameters were {'id': 2}

# Create an article.
>>> from datetime import datetime
>>> a = Article(pub_date=datetime.now(), headline='Django is cool',
...     content='Yeah.', reporter=r)
>>> a.save()

# Now the article is in the database.
>>> Article.objects.all()
[<Article: Django is cool>]
```

```
# Article objects get API access to related Reporter objects.
>>> r = a.reporter
>>> r.full_name
'John Smith'

# And vice versa: Reporter objects get API access to Article objects.
>>> r.article_set.all()
[<Article: Django is cool>]

# The API follows relationships as far as you need, performing efficient
# JOINs for you behind the scenes.
# This finds all articles by a reporter whose name starts with "John".
>>> Article.objects.filter(reporter__full_name__startswith="John")
[<Article: Django is cool>]

# Change an object by altering its attributes and calling save().
>>> r.full_name = 'Billy Goat'
>>> r.save()

# Delete an object with delete().
>>> r.delete()
```

2.1.4 A dynamic admin interface: it's not just scaffolding – it's the whole house

Once your models are defined, Django can automatically create a professional, production ready *administrative interface* – a Web site that lets authenticated users add, change and delete objects. It's as easy as registering your model in the admin site:

```
# In models.py...

from django.db import models

class Article(models.Model):
    pub_date = models.DateTimeField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)


# In admin.py in the same directory...

import models
from django.contrib import admin

admin.site.register(models.Article)
```

The philosophy here is that your site is edited by a staff, or a client, or maybe just you – and you don't want to have to deal with creating backend interfaces just to manage content.

One typical workflow in creating Django apps is to create models and get the admin sites up and running as fast as possible, so your staff (or clients) can start populating data. Then, develop the way data is presented to the public.

2.1.5 Design your URLs

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django encourages beautiful URL design and doesn't put any cruft in URLs, like `.php` or `.asp`.

To design URLs for an app, you create a Python module called a *URLconf*. A table of contents for your app, it contains a simple mapping between URL patterns and Python callback functions. URLconfs also serve to decouple URLs from Python code.

Here's what a URLconf might look like for the `Reporter/Article` example above:

```
from django.conf.urls import patterns, url, include

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

The code above maps URLs, as simple regular expressions, to the location of Python callback functions ("views"). The regular expressions use parenthesis to "capture" values from the URLs. When a user requests a page, Django runs through each pattern, in order, and stops at the first one that matches the requested URL. (If none of them matches, Django calls a special-case 404 view.) This is blazingly fast, because the regular expressions are compiled at load time.

Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function. Each view gets passed a request object – which contains request metadata – and the values captured in the regex.

For example, if a user requested the URL "/articles/2005/05/39323/", Django would call the function `news.views.article_detail(request, '2005', '05', '39323')`.

2.1.6 Write your views

Each view is responsible for doing one of two things: Returning an `HttpResponse` object containing the content for the requested page, or raising an exception such as `Http404`. The rest is up to you.

Generally, a view retrieves data according to the parameters, loads a template and renders the template with the retrieved data. Here's an example view for `year_archive` from above:

```
def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    return render_to_response('news/year_archive.html', {'year': year, 'article_list': a_list})
```

This example uses Django's *template system*, which has several powerful features but strives to stay simple enough for non-programmers to use.

2.1.7 Design your templates

The code above loads the `news/year_archive.html` template.

Django has a template search path, which allows you to minimize redundancy among templates. In your Django settings, you specify a list of directories to check for templates. If a template doesn't exist in the first directory, it checks the second, and so on.

Let's say the `news/year_archive.html` template was found. Here's what that might look like:

```
{% extends "base.html" %}

{% block title %}Articles for {{ year }}{% endblock %}

{% block content %}
<h1>Articles for {{ year }}</h1>
```

```
{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>By {{ article.reporter.full_name }}</p>
    <p>Published {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

Variables are surrounded by double-curly braces. `{{ article.headline }}` means "Output the value of the article's headline attribute." But dots aren't used only for attribute lookup: They also can do dictionary-key lookup, index lookup and function calls.

Note `{{ article.pub_date|date:"F j, Y" }}` uses a Unix-style "pipe" (the "|" character). This is called a template filter, and it's a way to filter the value of a variable. In this case, the date filter formats a Python datetime object in the given format (as found in PHP's date function; yes, there is one good idea in PHP).

You can chain together as many filters as you'd like. You can write custom filters. You can write custom template tags, which run custom Python code behind the scenes.

Finally, Django uses the concept of "template inheritance": That's what the `{% extends "base.html" %}` does. It means "First load the template called 'base', which has defined a bunch of blocks, and fill the blocks with the following blocks." In short, that lets you dramatically cut down on redundancy in templates: each template has to define only what's unique to that template.

Here's what the "base.html" template might look like:

```
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    <img src="sitelogo.png" alt="Logo" />
    {% block content %}{% endblock %}
</body>
</html>
```

Simplistically, it defines the look-and-feel of the site (with the site's logo), and provides "holes" for child templates to fill. This makes a site redesign as easy as changing a single file – the base template.

It also lets you create multiple versions of a site, with different base templates, while reusing child templates. Django's creators have used this technique to create strikingly different cell-phone editions of sites – simply by creating a new base template.

Note that you don't have to use Django's template system if you prefer another system. While Django's template system is particularly well-integrated with Django's model layer, nothing forces you to use it. For that matter, you don't have to use Django's database API, either. You can use another database abstraction layer, you can read XML files, you can read files off disk, or anything you want. Each piece of Django – models, views, templates – is decoupled from the next.

2.1.8 This is just the surface

This has been only a quick overview of Django's functionality. Some more useful features:

- A *caching framework* that integrates with memcached or other backends.
- A *syndication framework* that makes creating RSS and Atom feeds as easy as writing a small Python class.
- More sexy automatically-generated admin features – this overview barely scratched the surface.

The next obvious steps are for you to download Django, read *the tutorial* and join the community. Thanks for your interest!

2.2 Quick install guide

Before you can use Django, you'll need to get it installed. We have a *complete installation guide* that covers all the possibilities; this guide will guide you to a simple, minimal installation that'll work while you walk through the introduction.

2.2.1 Install Python

Being a Python Web framework, Django requires Python. It works with any Python version from 2.6.5 to 2.7 (due to backwards incompatibilities in Python 3.0, Django does not currently work with Python 3.0; see *the Django FAQ* for more information on supported Python versions and the 3.0 transition), these versions of Python include a lightweight database called SQLite so you won't need to set up a database just yet.

Get Python at http://www.python.org. If you're running Linux or Mac OS X, you probably already have it installed.

Django on Jython

If you use Jython (a Python implementation for the Java platform), you'll need to follow a few additional steps. See *Running Django on Jython* for details.

You can verify that Python is installed by typing `python` from your shell; you should see something like:

```
Python 2.6.6 (r266:84292, Dec 26 2010, 22:31:48)
[GCC 4.4.5] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2.2.2 Set up a database

This step is only necessary if you'd like to work with a "large" database engine like PostgreSQL, MySQL, or Oracle. To install such a database, consult the *database installation information*.

2.2.3 Remove any old versions of Django

If you are upgrading your installation of Django from a previous version, you will need to *uninstall the old Django version before installing the new version*.

2.2.4 Install Django

You've got three easy options to install Django:

- Install a version of Django *provided by your operating system distribution*. This is the quickest option for those who have operating systems that distribute Django.
- *Install an official release*. This is the best approach for users who want a stable version number and aren't concerned about running a slightly older version of Django.
- *Install the latest development version*. This is best for users who want the latest-and-greatest features and aren't afraid of running brand-new code.

Always refer to the documentation that corresponds to the version of Django you're using!

If you do either of the first two steps, keep an eye out for parts of the documentation marked **new in development version**. That phrase flags features that are only available in development versions of Django, and they likely won't work with an official release.

2.2.5 Verifying

To verify that Django can be seen by Python, type `python` from your shell. Then at the Python prompt, try to import Django:

```
>>> import django
>>> print(django.get_version())
1.4
```

2.2.6 That's it!

That's it – you can now *move onto the tutorial*.

2.3 Writing your first Django app, part 1

Let's learn by example.

Throughout this tutorial, we'll walk you through the creation of a basic poll application.

It'll consist of two parts:

- A public site that lets people view polls and vote in them.
- An admin site that lets you add, change and delete polls.

We'll assume you have *Django installed* already. You can tell Django is installed and which version by running the following command:

```
python -c "import django; print(django.get_version())"
```

You should see either the version of your Django installation or an error telling "No module named django". Check also that the version number matches the version of this tutorial. If they don't match, you can refer to the tutorial for your version of Django or update Django to the newest version.

See *How to install Django* for advice on how to remove older versions of Django and install a newer one.

Where to get help:

If you're having trouble going through this tutorial, please post a message to django-users or drop by #django on irc.freenode.net to chat with other Django users who might be able to help.

2.3.1 Creating a project

If this is your first time using Django, you'll have to take care of some initial setup. Namely, you'll need to auto-generate some code that establishes a Django *project* – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, `cd` into a directory where you'd like to store your code, then run the following command:

```
django-admin.py startproject mysite
```

This will create a `mysite` directory in your current directory.

Script name may differ in distribution packages

If you installed Django using a Linux distribution's package manager (e.g. apt-get or yum) `django-admin.py` may have been renamed to `django-admin`. You may continue through this documentation by omitting `.py` from each command.

Mac OS X permissions

If you're using Mac OS X, you may see the message "permission denied" when you try to run `django-admin.py startproject`. This is because, on Unix-based systems like OS X, a file must be marked as "executable" before it can be run as a program. To do this, open Terminal.app and navigate (using the `cd` command) to the directory where *django-admin.py* is installed, then run the command `sudo chmod +x django-admin.py`.

Note: You'll need to avoid naming projects after built-in Python or Django components. In particular, this means you should avoid using names like `django` (which will conflict with Django itself) or `test` (which conflicts with a built-in Python package).

---

django-admin.py should be on your system path if you installed Django via `python setup.py`. If it's not on your path, you can find it in `site-packages/django/bin`, where `site-packages` is a directory within your Python installation. Consider symlinking to *django-admin.py* from some place on your path, such as `/usr/local/bin`.

Where should this code live?

If your background is in PHP, you're probably used to putting code under the Web server's document root (in a place such as `/var/www`). With Django, you don't do that. It's not a good idea to put any of this Python code within your Web server's document root, because it risks the possibility that people may be able to view your code over the Web. That's not good for security.

Put your code in some directory **outside** of the document root, such as `/home/mycode`.

Let's look at what `startproject` created:

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        wsgi.py
```

Doesn't match what you see?

The default project layout recently changed. If you're seeing a "flat" layout (with no inner `mysite/` directory), you're probably using a version of Django that doesn't match this tutorial version. You'll want to either switch to the older tutorial or the newer Django version.

These files are:

- The outer `mysite/` directory is just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
- `manage.py`: A command-line utility that lets you interact with this Django project in various ways. You can read all the details about `manage.py` in *django-admin.py and manage.py*.
- The inner `mysite/` directory is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. `import mysite.settings`).
- `mysite/__init__.py`: An empty file that tells Python that this directory should be considered a Python package. (Read more about packages in the official Python docs if you're a Python beginner.)
- `mysite/settings.py`: Settings/configuration for this Django project. *Django settings* will tell you all about how settings work.
- `mysite/urls.py`: The URL declarations for this Django project; a "table of contents" of your Django-powered site. You can read more about URLs in *URL dispatcher*.
- `mysite/wsgi.py`: An entry-point for WSGI-compatible webservers to serve your project. See *How to deploy with WSGI* for more details.

The development server

Let's verify this worked. Change into the outer `mysite` directory, if you haven't already, and run the command `python manage.py runserver`. You'll see the following output on the command line:

```
Validating models...
0 errors found.

Django version 1.4, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

You've started the Django development server, a lightweight Web server written purely in Python. We've included this with Django so you can develop things rapidly, without having to deal with configuring a production server – such as Apache – until you're ready for production.

Now's a good time to note: DON'T use this server in anything resembling a production environment. It's intended only for use while developing. (We're in the business of making Web frameworks, not Web servers.)

Now that the server's running, visit http://127.0.0.1:8000/ with your Web browser. You'll see a "Welcome to Django" page, in pleasant, light-blue pastel. It worked!

Changing the port

By default, the `runserver` command starts the development server on the internal IP at port 8000.

If you want to change the server's port, pass it as a command-line argument. For instance, this command starts the server on port 8080:

```
python manage.py runserver 8080
```

If you want to change the server's IP, pass it along with the port. So to listen on all public IPs (useful if you want to show off your work on other computers), use:

```
python manage.py runserver 0.0.0.0:8000
```

Full docs for the development server can be found in the `runserver` reference.

Database setup

Now, edit `mysite/settings.py`. It's a normal Python module with module-level variables representing Django settings. Change the following keys in the `DATABASES 'default'` item to match your database connection settings.

- `ENGINE` – Either `'django.db.backends.postgresql_psycopg2'`, `'django.db.backends.mysql'`, `'django.db.backends.sqlite3'` or `'django.db.backends.oracle'`. Other backends are `also available`.
- `NAME` – The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, `NAME` should be the full absolute path, including filename, of that file. If the file doesn't exist, it will automatically be created when you synchronize the database for the first time (see below).

  When specifying the path, always use forward slashes, even on Windows (e.g. `C:/homes/user/mysite/sqlite3.db`).
- `USER` – Your database username (not used for SQLite).
- `PASSWORD` – Your database password (not used for SQLite).
- `HOST` – The host your database is on. Leave this as an empty string if your database server is on the same physical machine (not used for SQLite).

If you're new to databases, we recommend simply using SQLite by setting `ENGINE` to `'django.db.backends.sqlite3'` and `NAME` to the place where you'd like to store the database. SQLite is included in Python, so you won't need to install anything else to support your database.

Note: If you're using PostgreSQL or MySQL, make sure you've created a database by this point. Do that with "`CREATE DATABASE database_name;`" within your database's interactive prompt.

If you're using SQLite, you don't need to create anything beforehand - the database file will be created automatically when it is needed.

While you're editing `settings.py`, set `TIME_ZONE` to your time zone. The default value is the Central time zone in the U.S. (Chicago).

Also, note the `INSTALLED_APPS` setting toward the bottom of the file. That holds the names of all Django applications that are activated in this Django instance. Apps can be used in multiple projects, and you can package and distribute them for use by others in their projects.

By default, `INSTALLED_APPS` contains the following apps, all of which come with Django:

- `django.contrib.auth` – An authentication system.
- `django.contrib.contenttypes` – A framework for content types.
- `django.contrib.sessions` – A session framework.
- `django.contrib.sites` – A framework for managing multiple sites with one Django installation.
- `django.contrib.messages` – A messaging framework.
- `django.contrib.staticfiles` – A framework for managing static files.

These applications are included by default as a convenience for the common case.

Each of these applications makes use of at least one database table, though, so we need to create the tables in the database before we can use them. To do that, run the following command:

```
python manage.py syncdb
```

The `syncdb` command looks at the `INSTALLED_APPS` setting and creates any necessary database tables according to the database settings in your `settings.py` file. You'll see a message for each database table it creates, and you'll get a prompt asking you if you'd like to create a superuser account for the authentication system. Go ahead and do that.

If you're interested, run the command-line client for your database and type `\dt` (PostgreSQL), `SHOW TABLES;` (MySQL), or `.schema` (SQLite) to display the tables Django created.

For the minimalists

Like we said above, the default applications are included for the common case, but not everybody needs them. If you don't need any or all of them, feel free to comment-out or delete the appropriate line(s) from `INSTALLED_APPS` before running `syncdb`. The `syncdb` command will only create tables for apps in `INSTALLED_APPS`.

2.3.2 Creating models

Now that your environment – a "project" – is set up, you're set to start doing work.

Each application you write in Django consists of a Python package, somewhere on your Python path, that follows a certain convention. Django comes with a utility that automatically generates the basic directory structure of an app, so you can focus on writing code rather than creating directories.

Projects vs. apps

What's the difference between a project and an app? An app is a Web application that does something – e.g., a Weblog system, a database of public records or a simple poll app. A project is a collection of configuration and apps for a particular Web site. A project can contain multiple apps. An app can be in multiple projects.

Your apps can live anywhere on your Python path. In this tutorial, we'll create our poll app right next to your `manage.py` file so that it can be imported as its own top-level module, rather than a submodule of `mysite`.

To create your app, make sure you're in the same directory as `manage.py` and type this command:

```
python manage.py startapp polls
```

That'll create a directory `polls`, which is laid out like this:

```
polls/
    __init__.py
    models.py
    tests.py
    views.py
```

This directory structure will house the poll application.

The first step in writing a database Web app in Django is to define your models – essentially, your database layout, with additional metadata.

Philosophy

A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Django follows the *DRY Principle*. The goal is to define your data model in one place and automatically derive things from it.

In our simple poll app, we'll create two models: `Poll` and `Choice`. A `Poll` has a question and a publication date. A `Choice` has two fields: the text of the choice and a vote tally. Each `Choice` is associated with a `Poll`.

These concepts are represented by simple Python classes. Edit the `polls/models.py` file so it looks like this:

```
from django.db import models

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField()
```

The code is straightforward. Each model is represented by a class that subclasses `django.db.models.Model`. Each model has a number of class variables, each of which represents a database field in the model.

Each field is represented by an instance of a `Field` class – e.g., `CharField` for character fields and `DateTimeField` for datetimes. This tells Django what type of data each field holds.

The name of each `Field` instance (e.g. `question` or `pub_date`) is the field's name, in machine-friendly format. You'll use this value in your Python code, and your database will use it as the column name.

You can use an optional first positional argument to a `Field` to designate a human-readable name. That's used in a couple of introspective parts of Django, and it doubles as documentation. If this field isn't provided, Django will use the machine-readable name. In this example, we've only defined a human-readable name for `Poll.pub_date`. For all other fields in this model, the field's machine-readable name will suffice as its human-readable name.

Some `Field` classes have required elements. `CharField`, for example, requires that you give it a `max_length`. That's used not only in the database schema, but in validation, as we'll soon see.

Finally, note a relationship is defined, using `ForeignKey`. That tells Django each `Choice` is related to a single `Poll`. Django supports all the common database relationships: many-to-ones, many-to-manys and one-to-ones.

2.3.3 Activating models

That small bit of model code gives Django a lot of information. With it, Django is able to:

- Create a database schema (`CREATE TABLE` statements) for this app.
- Create a Python database-access API for accessing `Poll` and `Choice` objects.

But first we need to tell our project that the `polls` app is installed.

Philosophy

Django apps are "pluggable": You can use an app in multiple projects, and you can distribute apps, because they don't have to be tied to a given Django installation.

Edit the `settings.py` file again, and change the `INSTALLED_APPS` setting to include the string `'polls'`. So it'll look like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
```

```
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'polls',
)
```

Now Django knows to include the `polls` app. Let's run another command:

```
python manage.py sql polls
```

You should see something similar to the following (the `CREATE TABLE` SQL statements for the polls app):

```
BEGIN;
CREATE TABLE "polls_poll" (
    "id" serial NOT NULL PRIMARY KEY,
    "question" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id") DEFERRABLE INITIALLY DEFERRED,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
COMMIT;
```

Note the following:

- The exact output will vary depending on the database you are using.
- Table names are automatically generated by combining the name of the app (`polls`) and the lowercase name of the model – `poll` and `choice`. (You can override this behavior.)
- Primary keys (IDs) are added automatically. (You can override this, too.)
- By convention, Django appends `"_id"` to the foreign key field name. (Yes, you can override this, as well.)
- The foreign key relationship is made explicit by a `REFERENCES` statement.
- It's tailored to the database you're using, so database-specific field types such as `auto_increment` (MySQL), `serial` (PostgreSQL), or `integer primary key` (SQLite) are handled for you automatically. Same goes for quoting of field names – e.g., using double quotes or single quotes. The author of this tutorial runs PostgreSQL, so the example output is in PostgreSQL syntax.
- The `sql` command doesn't actually run the SQL in your database - it just prints it to the screen so that you can see what SQL Django thinks is required. If you wanted to, you could copy and paste this SQL into your database prompt. However, as we will see shortly, Django provides an easier way of committing the SQL to the database.

If you're interested, also run the following commands:

- `python manage.py validate` – Checks for any errors in the construction of your models.
- `python manage.py sqlcustom polls` – Outputs any *custom SQL statements* (such as table modifications or constraints) that are defined for the application.
- `python manage.py sqlclear polls` – Outputs the necessary `DROP TABLE` statements for this app, according to which tables already exist in your database (if any).
- `python manage.py sqlindexes polls` – Outputs the `CREATE INDEX` statements for this app.
- `python manage.py sqlall polls` – A combination of all the SQL from the `sql`, `sqlcustom`, and `sqlindexes` commands.

Looking at the output of those commands can help you understand what's actually happening under the hood.

Now, run `syncdb` again to create those model tables in your database:

```
python manage.py syncdb
```

The `syncdb` command runs the SQL from `sqlall` on your database for all apps in `INSTALLED_APPS` that don't already exist in your database. This creates all the tables, initial data and indexes for any apps you've added to your project since the last time you ran syncdb. `syncdb` can be called as often as you like, and it will only ever create the tables that don't exist.

Read the *django-admin.py documentation* for full information on what the `manage.py` utility can do.

2.3.4 Playing with the API

Now, let's hop into the interactive Python shell and play around with the free API Django gives you. To invoke the Python shell, use this command:

```
python manage.py shell
```

We're using this instead of simply typing "python", because `manage.py` sets the `DJANGO_SETTINGS_MODULE` environment variable, which gives Django the Python import path to your `settings.py` file.

Bypassing manage.py

If you'd rather not use `manage.py`, no problem. Just set the `DJANGO_SETTINGS_MODULE` environment variable to `mysite.settings` and run `python` from the same directory `manage.py` is in (or ensure that directory is on the Python path, so that `import mysite` works).

For more information on all of this, see the *django-admin.py documentation*.

Once you're in the shell, explore the *database API*:

```
>>> from polls.models import Poll, Choice   # Import the model classes we just wrote.

# No polls are in the system yet.
>>> Poll.objects.all()
[]

# Create a new Poll.
# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.
>>> from django.utils import timezone
>>> p = Poll(question="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.
>>> p.save()

# Now it has an ID. Note that this might say "1L" instead of "1", depending
# on which database you're using. That's no biggie; it just means your
# database backend prefers to return integers as Python long integer
# objects.
>>> p.id
1

# Access database columns via Python attributes.
>>> p.question
```

```
"What's new?"
>>> p.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().
>>> p.question = "What's up?"
>>> p.save()

# objects.all() displays all the polls in the database.
>>> Poll.objects.all()
[<Poll: Poll object>]
```

Wait a minute. `<Poll: Poll object>` is, utterly, an unhelpful representation of this object. Let's fix that by editing the polls model (in the `polls/models.py` file) and adding a `__unicode__()` method to both `Poll` and `Choice`:

```
class Poll(models.Model):
    # ...
    def __unicode__(self):
        return self.question

class Choice(models.Model):
    # ...
    def __unicode__(self):
        return self.choice_text
```

It's important to add `__unicode__()` methods to your models, not only for your own sanity when dealing with the interactive prompt, but also because objects' representations are used throughout Django's automatically-generated admin.

Why `__unicode__()` and not `__str__()`?

If you're familiar with Python, you might be in the habit of adding `__str__()` methods to your classes, not `__unicode__()` methods. We use `__unicode__()` here because Django models deal with Unicode by default. All data stored in your database is converted to Unicode when it's returned.

Django models have a default `__str__()` method that calls `__unicode__()` and converts the result to a UTF-8 bytestring. This means that `unicode(p)` will return a Unicode string, and `str(p)` will return a normal string, with characters encoded as UTF-8.

If all of this is gibberish to you, just remember to add `__unicode__()` methods to your models. With any luck, things should Just Work for you.

Note these are normal Python methods. Let's add a custom method, just for demonstration:

```
import datetime
from django.utils import timezone
# ...
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Note the addition of `import datetime` and `from django.utils import timezone`, to reference Python's standard `datetime` module and Django's time-zone-related utilities in `django.utils.timezone`, respectively. If you aren't familiar with time zone handling in Python, you can learn more in the *time zone support docs*.

Save these changes and start a new Python interactive shell by running `python manage.py shell` again:

```
>>> from polls.models import Poll, Choice

# Make sure our __unicode__() addition worked.
>>> Poll.objects.all()
[<Poll: What's up?>]

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.
>>> Poll.objects.filter(id=1)
[<Poll: What's up?>]
>>> Poll.objects.filter(question__startswith='What')
[<Poll: What's up?>]

# Get the poll whose year is 2012.
>>> Poll.objects.get(pub_date__year=2012)
<Poll: What's up?>

>>> Poll.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Poll matching query does not exist. Lookup parameters were {'id': 2}

# Lookup by a primary key is the most common case, so Django provides a
# shortcut for primary-key exact lookups.
# The following is identical to Poll.objects.get(id=1).
>>> Poll.objects.get(pk=1)
<Poll: What's up?>

# Make sure our custom method worked.
>>> p = Poll.objects.get(pk=1)
>>> p.was_published_recently()
True

# Give the Poll a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a poll's choices) which can be accessed via the API.
>>> p = Poll.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> p.choice_set.all()
[]

# Create three choices.
>>> p.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> p.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = p.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Poll objects.
>>> c.poll
<Poll: What's up?>

# And vice versa: Poll objects get access to Choice objects.
>>> p.choice_set.all()
```

```
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> p.choice_set.count()
3

# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any poll whose pub_date is in 2012.
>>> Choice.objects.filter(poll__pub_date__year=2012)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

# Let's delete one of the choices. Use delete() for that.
>>> c = p.choice_set.filter(choice_text__startswith='Just hacking')
>>> c.delete()
```

For more information on model relations, see *Accessing related objects*. For more on how to use double underscores to perform field lookups via the API, see *Field lookups*. For full details on the database API, see our *Database API reference*.

When you're comfortable with the API, read *part 2 of this tutorial* to get Django's automatic admin working.

2.4 Writing your first Django app, part 2

This tutorial begins where *Tutorial 1* left off. We're continuing the Web-poll application and will focus on Django's automatically-generated admin site.

Philosophy

Generating admin sites for your staff or clients to add, change and delete content is tedious work that doesn't require much creativity. For that reason, Django entirely automates creation of admin interfaces for models.

Django was written in a newsroom environment, with a very clear separation between "content publishers" and the "public" site. Site managers use the system to add news stories, events, sports scores, etc., and that content is displayed on the public site. Django solves the problem of creating a unified interface for site administrators to edit content.

The admin isn't intended to be used by site visitors. It's for site managers.

2.4.1 Activate the admin site

The Django admin site is not activated by default – it's an opt-in thing. To activate the admin site for your installation, do these three things:

- Uncomment `"django.contrib.admin"` in the `INSTALLED_APPS` setting.
- Run `python manage.py syncdb`. Since you have added a new application to `INSTALLED_APPS`, the database tables need to be updated.
- Edit your `mysite/urls.py` file and uncomment the lines that reference the admin – there are three lines in total to uncomment. This file is a URLconf; we'll dig into URLconfs in the next tutorial. For now, all you need to know is that it maps URL roots to applications. In the end, you should have a `urls.py` file that looks like this:

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
```

```
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', '{{ project_name }}.views.home', name='home'),
    # url(r'^{{ project_name }}/', include('{{ project_name }}.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

(The bold lines are the ones that needed to be uncommented.)

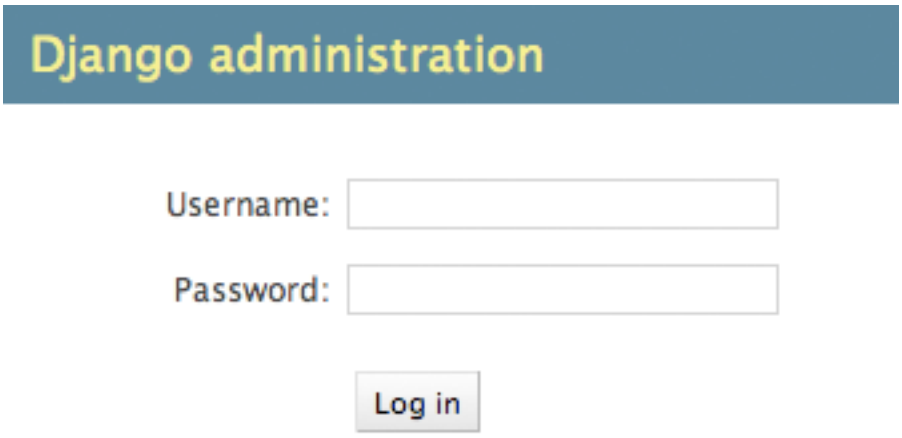
2.4.2 Start the development server

Let's start the development server and explore the admin site.

Recall from Tutorial 1 that you start the development server like so:

```
python manage.py runserver
```

Now, open a Web browser and go to "/admin/" on your local domain – e.g., http://127.0.0.1:8000/admin/. You should see the admin's login screen:



Doesn't match what you see?

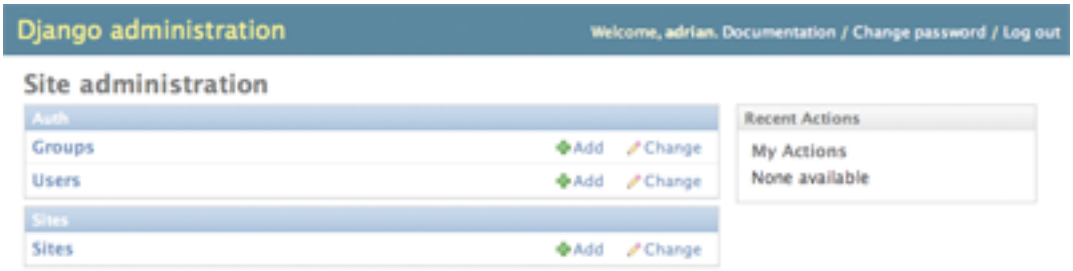
If at this point, instead of the above login page, you get an error page reporting something like:

```
ImportError at /admin/
cannot import name patterns
...
```

then you're probably using a version of Django that doesn't match this tutorial version. You'll want to either switch to the older tutorial or the newer Django version.

2.4.3 Enter the admin site

Now, try logging in. (You created a superuser account in the first part of this tutorial, remember? If you didn't create one or forgot the password you can *create another one*.) You should see the Django admin index page:



You should see a few types of editable content, including groups, users and sites. These are core features Django ships with by default.

2.4.4 Make the poll app modifiable in the admin

But where's our poll app? It's not displayed on the admin index page.

Just one thing to do: We need to tell the admin that `Poll` objects have an admin interface. To do this, create a file called `admin.py` in your `polls` directory, and edit it to look like this:

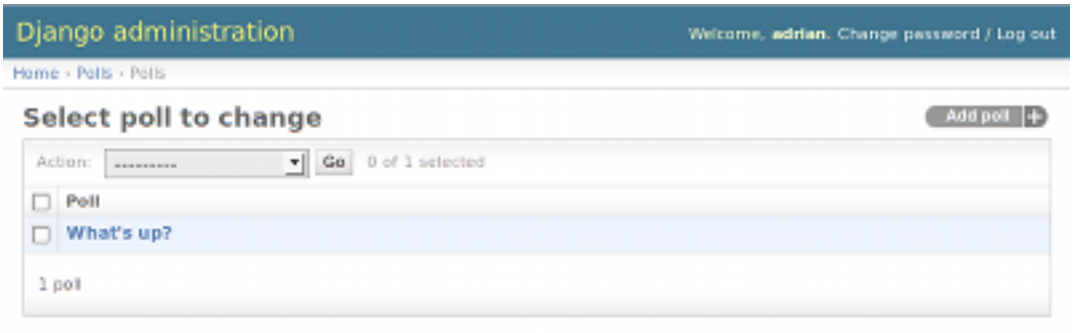
```
from django.contrib import admin
from polls.models import Poll

admin.site.register(Poll)
```

You'll need to restart the development server to see your changes. Normally, the server auto-reloads code every time you modify a file, but the action of creating a new file doesn't trigger the auto-reloading logic.

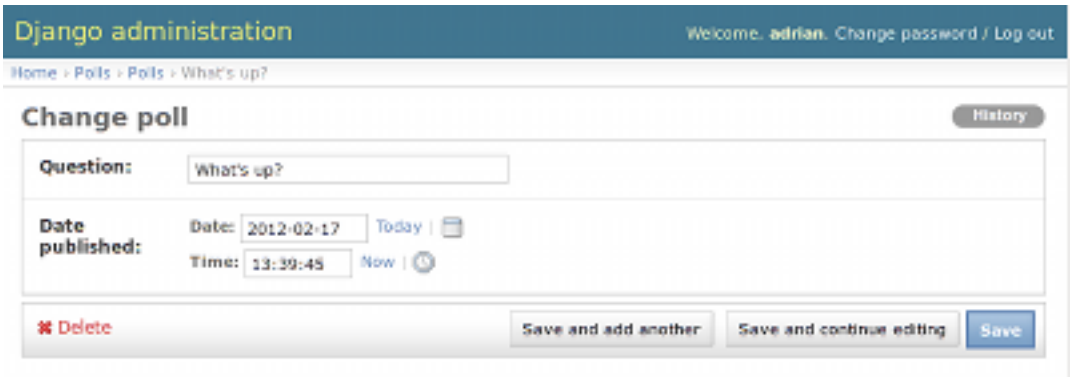
2.4.5 Explore the free admin functionality

Now that we've registered `Poll`, Django knows that it should be displayed on the admin index page:

Click "Polls." Now you're at the "change list" page for polls. This page displays all the polls in the database and lets you choose one to change it. There's the "What's up?" poll we created in the first tutorial:



Click the "What's up?" poll to edit it:



Things to note here:

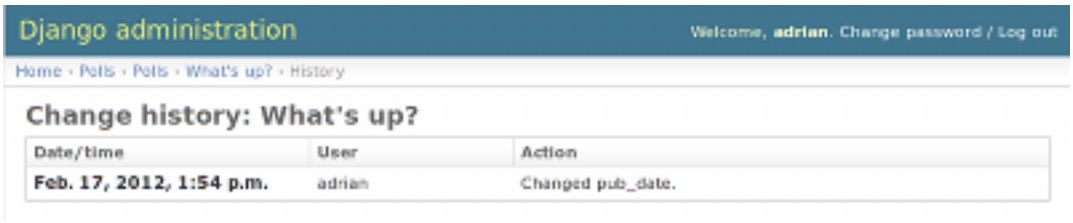
- The form is automatically generated from the Poll model.
- The different model field types (`DateTimeField`, `CharField`) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
- Each `DateTimeField` gets free JavaScript shortcuts. Dates get a "Today" shortcut and calendar popup, and times get a "Now" shortcut and a convenient popup that lists commonly entered times.

The bottom part of the page gives you a couple of options:

- Save – Saves changes and returns to the change-list page for this type of object.
- Save and continue editing – Saves changes and reloads the admin page for this object.
- Save and add another – Saves changes and loads a new, blank form for this type of object.
- Delete – Displays a delete confirmation page.

If the value of "Date published" doesn't match the time when you created the poll in Tutorial 1, it probably means you forgot to set the correct value for the `TIME_ZONE` setting. Change it, reload the page and check that the correct value appears.

Change the "Date published" by clicking the "Today" and "Now" shortcuts. Then click "Save and continue editing." Then click "History" in the upper right. You'll see a page listing all changes made to this object via the Django admin, with the timestamp and username of the person who made the change:



2.4.6 Customize the admin form

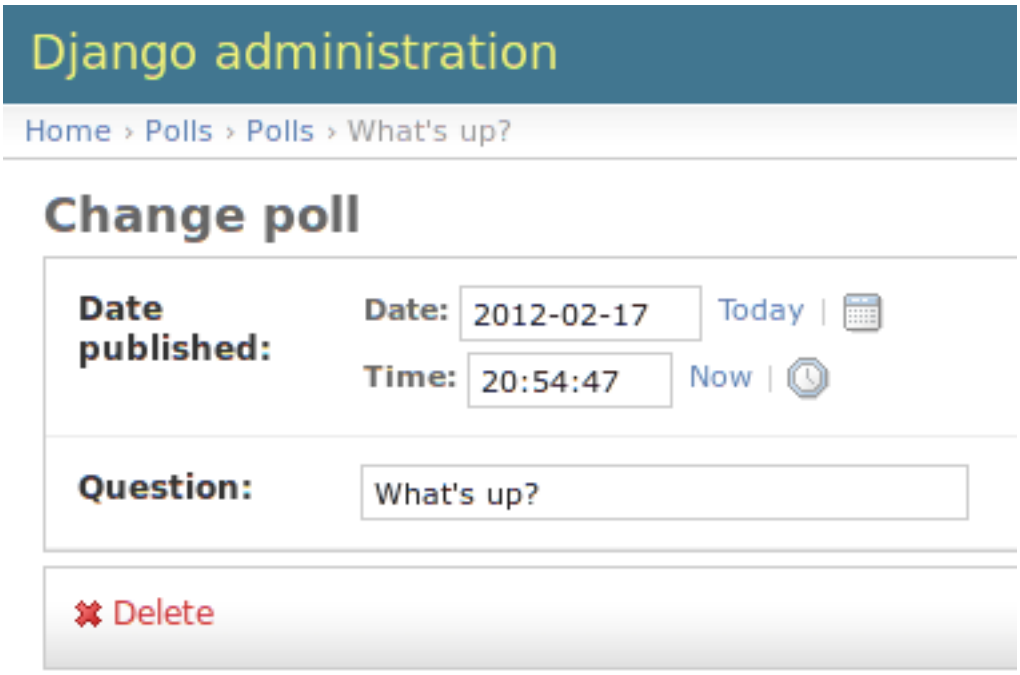
Take a few minutes to marvel at all the code you didn't have to write. By registering the Poll model with `admin.site.register(Poll)`, Django was able to construct a default form representation. Often, you'll want to customize how the admin form looks and works. You'll do this by telling Django the options you want when you register the object.

Let's see how this works by re-ordering the fields on the edit form. Replace the `admin.site.register(Poll)` line with:

```
class PollAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question']

admin.site.register(Poll, PollAdmin)
```

You'll follow this pattern – create a model admin object, then pass it as the second argument to `admin.site.register()` – any time you need to change the admin options for an object.

This particular change above makes the "Publication date" come before the "Question" field:

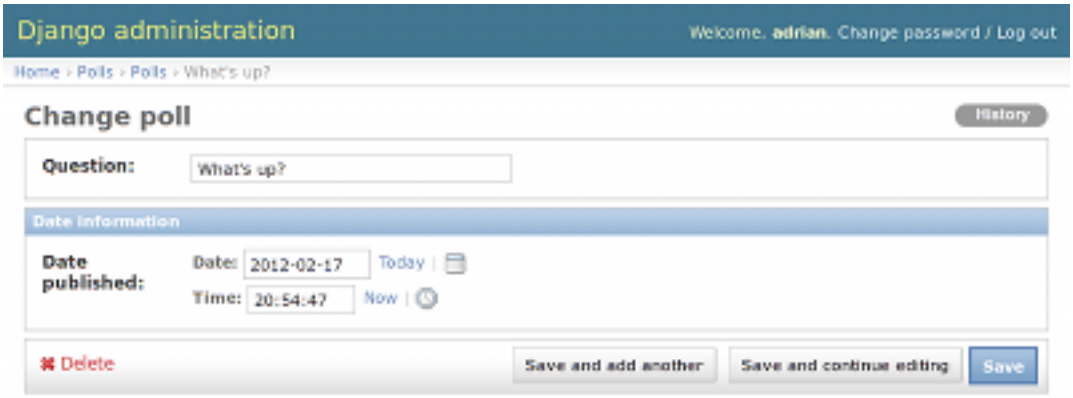


This isn't impressive with only two fields, but for admin forms with dozens of fields, choosing an intuitive order is an important usability detail.

And speaking of forms with dozens of fields, you might want to split the form up into fieldsets:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Poll, PollAdmin)
```

The first element of each tuple in `fieldsets` is the title of the fieldset. Here's what our form looks like now:



You can assign arbitrary HTML classes to each fieldset. Django provides a `"collapse"` class that displays a particular fieldset initially collapsed. This is useful when you have a long form that contains a number of fields that aren't commonly used:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
```

2.4.7 Adding related objects

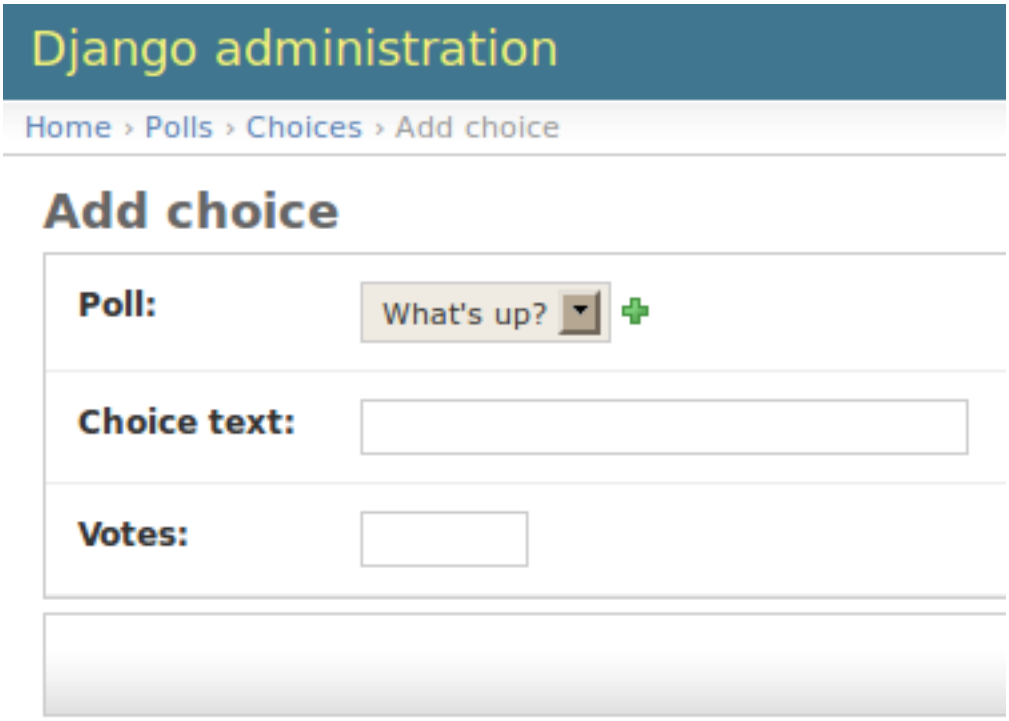
OK, we have our Poll admin page. But a `Poll` has multiple `Choices`, and the admin page doesn't display choices.

Yet.

There are two ways to solve this problem. The first is to register `Choice` with the admin just as we did with `Poll`. That's easy:

```
from polls.models import Choice

admin.site.register(Choice)
```

Now "Choices" is an available option in the Django admin. The "Add choice" form looks like this:



In that form, the "Poll" field is a select box containing every poll in the database. Django knows that a `ForeignKey` should be represented in the admin as a `<select>` box. In our case, only one poll exists at this point.

Also note the "Add Another" link next to "Poll." Every object with a `ForeignKey` relationship to another gets this for free. When you click "Add Another," you'll get a popup window with the "Add poll" form. If you add a poll in that window and click "Save," Django will save the poll to the database and dynamically add it as the selected choice on the "Add choice" form you're looking at.

But, really, this is an inefficient way of adding `Choice` objects to the system. It'd be better if you could add a bunch of Choices directly when you create the `Poll` object. Let's make that happen.

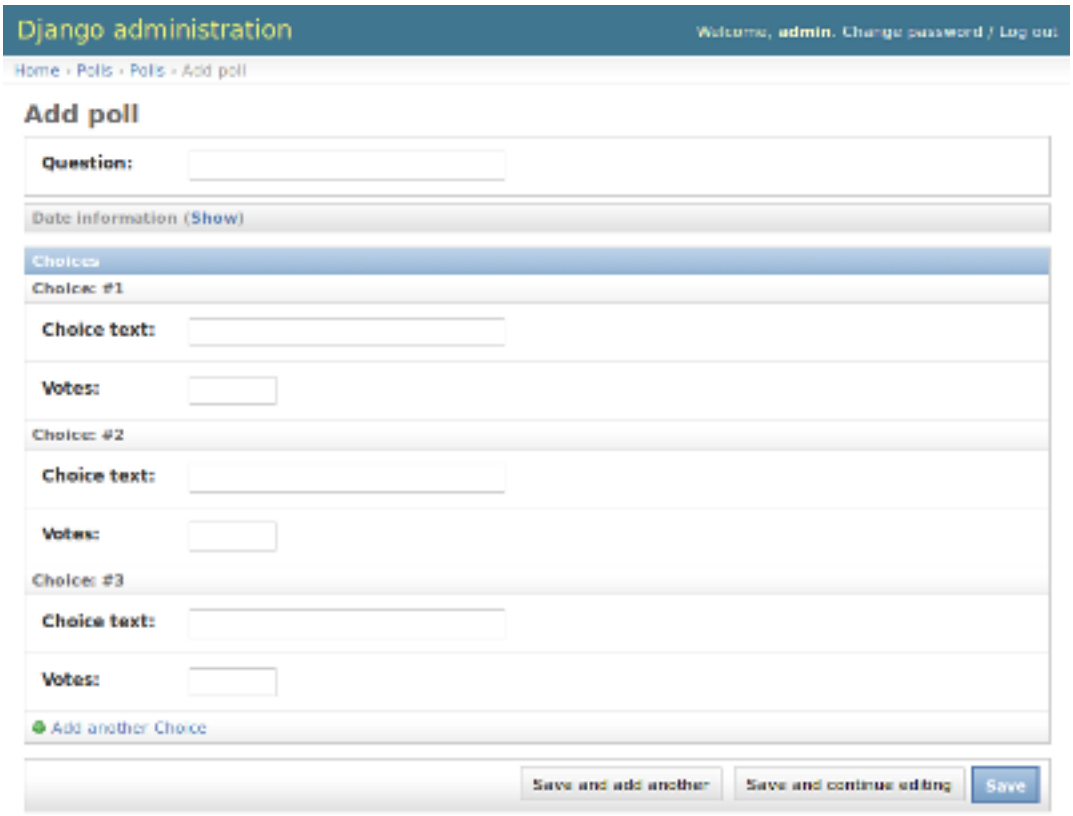
Remove the `register()` call for the `Choice` model. Then, edit the `Poll` registration code to read:

```
from django.contrib import admin
from polls.models import Choice, Poll

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3
```

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,               {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

This tells Django: "`Choice` objects are edited on the `Poll` admin page. By default, provide enough fields for 3 choices."

Load the "Add poll" page to see how that looks, you may need to restart your development server:

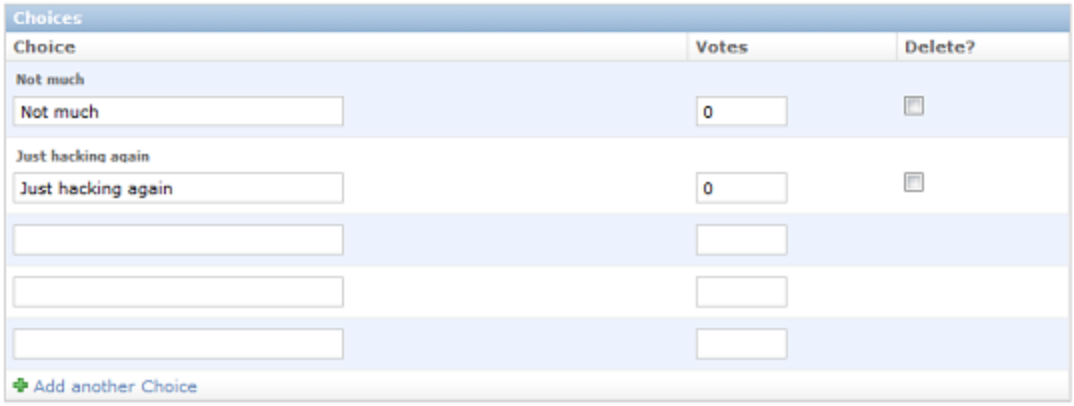


It works like this: There are three slots for related Choices – as specified by `extra` – and each time you come back to the "Change" page for an already-created object, you get another three extra slots.

One small problem, though. It takes a lot of screen space to display all the fields for entering related `Choice` objects. For that reason, Django offers a tabular way of displaying inline related objects; you just need to change the `ChoiceInline` declaration to read:

```
class ChoiceInline(admin.TabularInline):
    #...
```

With that `TabularInline` (instead of `StackedInline`), the related objects are displayed in a more compact, table-based format:

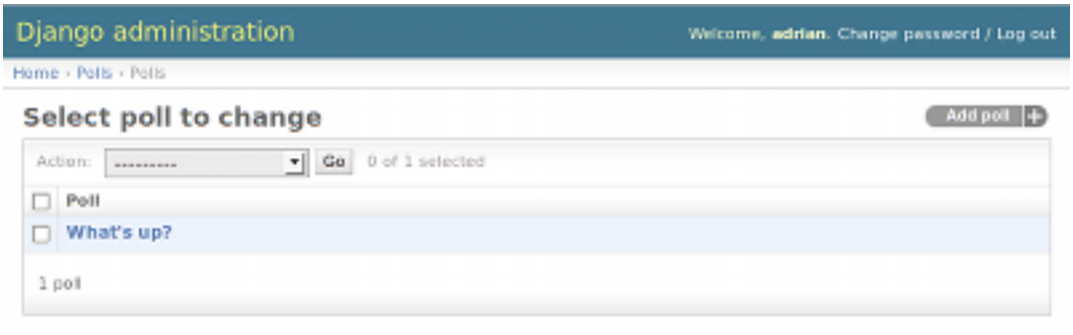


Note that there is an extra "Delete?" column that allows removing rows added using the "Add Another Choice" button and rows that have already been saved.

2.4.8 Customize the admin change list

Now that the Poll admin page is looking good, let's make some tweaks to the "change list" page – the one that displays all the polls in the system.

Here's what it looks like at this point:



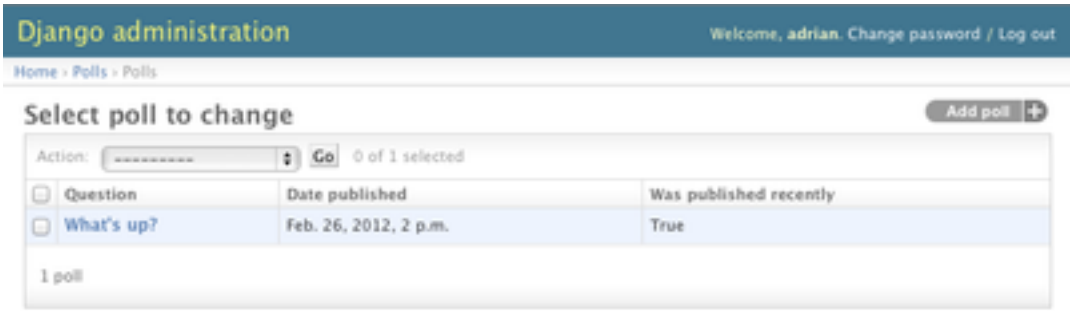
By default, Django displays the `str()` of each object. But sometimes it'd be more helpful if we could display individual fields. To do that, use the `list_display` admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date')
```

Just for good measure, let's also include the `was_published_recently` custom method from Tutorial 1:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date', 'was_published_recently')
```

Now the poll change list page looks like this:



You can click on the column headers to sort by those values – except in the case of the `was_published_recently` header, because sorting by the output of an arbitrary method is not supported. Also note that the column header for `was_published_recently` is, by default, the name of the method (with underscores replaced with spaces), and that each line contains the string representation of the output.

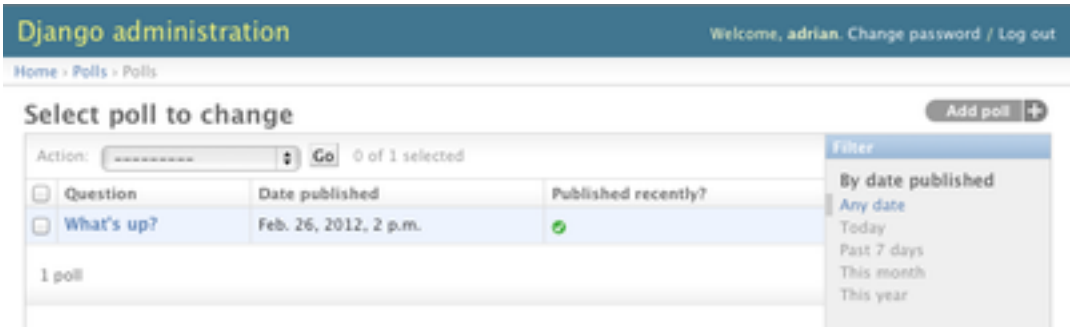
You can improve that by giving that method (in `models.py`) a few attributes, as follows:

```
class Poll(models.Model):
    # ...
    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
    was_published_recently.short_description = 'Published recently?'
```

Edit your admin.py file again and add an improvement to the Poll change list page: Filters. Add the following line to `PollAdmin`:

```
list_filter = ['pub_date']
```

That adds a "Filter" sidebar that lets people filter the change list by the `pub_date` field:



The type of filter displayed depends on the type of field you're filtering on. Because `pub_date` is a `DateTimeField`, Django knows to give appropriate filter options: "Any date," "Today," "Past 7 days," "This month," "This year."

This is shaping up well. Let's add some search capability:

```
search_fields = ['question']
```

That adds a search box at the top of the change list. When somebody enters search terms, Django will search the `question` field. You can use as many fields as you'd like – although because it uses a `LIKE` query behind the scenes, keep it reasonable, to keep your database happy.

Finally, because `Poll` objects have dates, it'd be convenient to be able to drill down by date. Add this line:

```
date_hierarchy = 'pub_date'
```

That adds hierarchical navigation, by date, to the top of the change list page. At top level, it displays all available years. Then it drills down to months and, ultimately, days.

Now's also a good time to note that change lists give you free pagination. The default is to display 100 items per page. Change-list pagination, search boxes, filters, date-hierarchies and column-header-ordering all work together like you think they should.

2.4.9 Customize the admin look and feel

Clearly, having "Django administration" at the top of each admin page is ridiculous. It's just placeholder text.

That's easy to change, though, using Django's template system. The Django admin is powered by Django itself, and its interfaces use Django's own template system.

Open your settings file (`mysite/settings.py`, remember) and look at the `TEMPLATE_DIRS` setting. `TEMPLATE_DIRS` is a tuple of filesystem directories to check when loading Django templates. It's a search path.

By default, `TEMPLATE_DIRS` is empty. So, let's add a line to it, to tell Django where our templates live:

```
TEMPLATE_DIRS = (
    '/home/my_username/mytemplates', # Change this to your own directory.
)
```

Now copy the template `admin/base_site.html` from within the default Django admin template directory in the source code of Django itself (`django/contrib/admin/templates`) into an `admin` subdirectory of whichever directory you're using in `TEMPLATE_DIRS`. For example, if your `TEMPLATE_DIRS` includes `'/home/my_username/mytemplates'`, as above, then copy `django/contrib/admin/templates/admin/base_site.html` to `/home/my_username/mytemplates/admin/base_site.html`. Don't forget that `admin` subdirectory.

Where are the Django source files?

If you have difficulty finding where the Django source files are located on your system, run the following command:

```
python -c "
import sys
sys.path = sys.path[1:]
import django
print(django.__path__)"
```

Then, just edit the file and replace the generic Django text with your own site's name as you see fit.

This template file contains lots of text like `{% block branding %}` and `{{ title }}`. The `{%` and `{{` tags are part of Django's template language. When Django renders `admin/base_site.html`, this template language will be evaluated to produce the final HTML page. Don't worry if you can't make any sense of the template right now – we'll delve into Django's templating language in Tutorial 3.

Note that any of Django's default admin templates can be overridden. To override a template, just do the same thing you did with `base_site.html` – copy it from the default directory into your custom directory, and make changes.

Astute readers will ask: But if `TEMPLATE_DIRS` was empty by default, how was Django finding the default admin templates? The answer is that, by default, Django automatically looks for a `templates/` subdirectory within each app package, for use as a fallback. See the *template loader documentation* for full information.

2.4.10 Customize the admin index page

On a similar note, you might want to customize the look and feel of the Django admin index page.

By default, it displays all the apps in `INSTALLED_APPS` that have been registered with the admin application, in alphabetical order. You may want to make significant changes to the layout. After all, the index is probably the most important page of the admin, and it should be easy to use.

The template to customize is `admin/index.html`. (Do the same as with `admin/base_site.html` in the previous section – copy it from the default directory to your custom template directory.) Edit the file, and you'll see it uses a template variable called `app_list`. That variable contains every installed Django app. Instead of using that, you can hard-code links to object-specific admin pages in whatever way you think is best. Again, don't worry if you can't understand the template language – we'll cover that in more detail in Tutorial 3.

When you're comfortable with the admin site, read *part 3 of this tutorial* to start working on public poll views.

2.5 Writing your first Django app, part 3

This tutorial begins where *Tutorial 2* left off. We're continuing the Web-poll application and will focus on creating the public interface – "views."

2.5.1 Philosophy

A view is a "type" of Web page in your Django application that generally serves a specific function and has a specific template. For example, in a Weblog application, you might have the following views:

- Blog homepage – displays the latest few entries.
- Entry "detail" page – permalink page for a single entry.
- Year-based archive page – displays all months with entries in the given year.
- Month-based archive page – displays all days with entries in the given month.
- Day-based archive page – displays all entries in the given day.
- Comment action – handles posting comments to a given entry.

In our poll application, we'll have the following four views:

- Poll "index" page – displays the latest few polls.
- Poll "detail" page – displays a poll question, with no results but with a form to vote.
- Poll "results" page – displays results for a particular poll.
- Vote action – handles voting for a particular choice in a particular poll.

In Django, each view is represented by a simple Python function.

2.5.2 Design your URLs

The first step of writing views is to design your URL structure. You do this by creating a Python module, called a URLconf. URLconfs are how Django associates a given URL with given Python code.

When a user requests a Django-powered page, the system looks at the `ROOT_URLCONF` setting, which contains a string in Python dotted syntax. Django loads that module and looks for a module-level variable called `urlpatterns`, which is a sequence of tuples in the following format:

```
(regular expression, Python callback function [, optional dictionary])
```

Django starts at the first regular expression and makes its way down the list, comparing the requested URL against each regular expression until it finds one that matches.

When it finds a match, Django calls the Python callback function, with an `HttpRequest` object as the first argument, any "captured" values from the regular expression as keyword arguments, and, optionally, arbitrary keyword arguments from the dictionary (an optional third item in the tuple).

For more on `HttpRequest` objects, see the *Request and response objects*. For more details on URLconfs, see the *URL dispatcher*.

When you ran `django-admin.py startproject mysite` at the beginning of Tutorial 1, it created a default URLconf in `mysite/urls.py`. It also automatically set your `ROOT_URLCONF` setting (in `settings.py`) to point at that file:

```
ROOT_URLCONF = 'mysite.urls'
```

Time for an example. Edit `mysite/urls.py` so it looks like this:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/$', 'polls.views.index'),
    url(r'^polls/(?P<poll_id>\d+)/$', 'polls.views.detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'polls.views.results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'polls.views.vote'),
    url(r'^admin/', include(admin.site.urls)),
)
```

This is worth a review. When somebody requests a page from your Web site – say, "/polls/23/", Django will load this Python module, because it's pointed to by the `ROOT_URLCONF` setting. It finds the variable named `urlpatterns` and traverses the regular expressions in order. When it finds a regular expression that matches – `r'^polls/(?P<poll_id>\d+)/$'` – it loads the function `detail()` from `polls/views.py`. Finally, it calls that `detail()` function like so:

```
detail(request=<HttpRequest object>, poll_id='23')
```

The `poll_id='23'` part comes from `(?P<poll_id>\d+)`. Using parentheses around a pattern "captures" the text matched by that pattern and sends it as an argument to the view function; the `?P<poll_id>` defines the name that will be used to identify the matched pattern; and `\d+` is a regular expression to match a sequence of digits (i.e., a number).

Because the URL patterns are regular expressions, there really is no limit on what you can do with them. And there's no need to add URL cruft such as `.php` – unless you have a sick sense of humor, in which case you can do something like this:

```
(r'^polls/latest\.php$', 'polls.views.index'),
```

But, don't do that. It's silly.

Note that these regular expressions do not search GET and POST parameters, or the domain name. For example, in a request to `http://www.example.com/myapp/`, the URLconf will look for `myapp/`. In a request to `http://www.example.com/myapp/?page=3`, the URLconf will look for `myapp/`.

If you need help with regular expressions, see Wikipedia's entry and the documentation of the `re` module. Also, the O'Reilly book "Mastering Regular Expressions" by Jeffrey Friedl is fantastic.

Finally, a performance note: these regular expressions are compiled the first time the URLconf module is loaded. They're super fast.

2.5.3 Write your first view

Well, we haven't created any views yet – we just have the URLconf. But let's make sure Django is following the URLconf properly.

Fire up the Django development Web server:

```
python manage.py runserver
```

Now go to "http://localhost:8000/polls/" on your domain in your Web browser. You should get a pleasantly-colored error page with the following message:

```
ViewDoesNotExist at /polls/

Could not import polls.views.index. View does not exist in module polls.views.
```

This error happened because you haven't written a function `index()` in the module `polls/views.py`.

Try "/polls/23/", "/polls/23/results/" and "/polls/23/vote/". The error messages tell you which view Django tried (and failed to find, because you haven't written any views yet).

Time to write the first view. Open the file `polls/views.py` and put the following Python code in it:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world. You're at the poll index.")
```

This is the simplest view possible. Go to "/polls/" in your browser, and you should see your text.

Now lets add a few more views. These views are slightly different, because they take an argument (which, remember, is passed in from whatever was captured by the regular expression in the URLconf):

```
def detail(request, poll_id):
    return HttpResponse("You're looking at poll %s." % poll_id)

def results(request, poll_id):
    return HttpResponse("You're looking at the results of poll %s." % poll_id)

def vote(request, poll_id):
    return HttpResponse("You're voting on poll %s." % poll_id)
```

Take a look in your browser, at "/polls/34/". It'll run the *detail()* method and display whatever ID you provide in the URL. Try "/polls/34/results/" and "/polls/34/vote/" too – these will display the placeholder results and voting pages.

2.5.4 Write views that actually do something

Each view is responsible for doing one of two things: Returning an `HttpResponse` object containing the content for the requested page, or raising an exception such as `Http404`. The rest is up to you.

Your view can read records from a database, or not. It can use a template system such as Django's – or a third-party Python template system – or not. It can generate a PDF file, output XML, create a ZIP file on the fly, anything you want, using whatever Python libraries you want.

All Django wants is that `HttpResponse`. Or an exception.

Because it's convenient, let's use Django's own database API, which we covered in *Tutorial 1*. Here's one stab at the `index()` view, which displays the latest 5 poll questions in the system, separated by commas, according to publication date:

```
from polls.models import Poll
from django.http import HttpResponse

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    output = ', '.join([p.question for p in latest_poll_list])
    return HttpResponse(output)
```

There's a problem here, though: The page's design is hard-coded in the view. If you want to change the way the page looks, you'll have to edit this Python code. So let's use Django's template system to separate the design from Python:

```
from django.template import Context, loader
from polls.models import Poll
from django.http import HttpResponse

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    t = loader.get_template('polls/index.html')
    c = Context({
        'latest_poll_list': latest_poll_list,
    })
    return HttpResponse(t.render(c))
```

That code loads the template called "polls/index.html" and passes it a context. The context is a dictionary mapping template variable names to Python objects.

Reload the page. Now you'll see an error:

```
TemplateDoesNotExist at /polls/
polls/index.html
```

Ah. There's no template yet. First, create a directory, somewhere on your filesystem, whose contents Django can access. (Django runs as whatever user your server runs.) Don't put them under your document root, though. You probably shouldn't make them public, just for security's sake. Then edit `TEMPLATE_DIRS` in your `settings.py` to tell Django where it can find templates – just as you did in the "Customize the admin look and feel" section of Tutorial 2.

When you've done that, create a directory `polls` in your template directory. Within that, create a file called `index.html`. Note that our `loader.get_template('polls/index.html')` code from above maps to "[template_directory]/polls/index.html" on the filesystem.

Put the following code in that template:

```
{% if latest_poll_list %}
    <ul>
    {% for poll in latest_poll_list %}
        <li><a href="/polls/{{ poll.id }}/">{{ poll.question }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>No polls are available.</p>
{% endif %}
```

Load the page in your Web browser, and you should see a bulleted-list containing the "What's up" poll from Tutorial 1. The link points to the poll's detail page.

A shortcut: render_to_response()

It's a very common idiom to load a template, fill a context and return an `HttpResponse` object with the result of the rendered template. Django provides a shortcut. Here's the full `index()` view, rewritten:

```
from django.shortcuts import render_to_response
from polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    return render_to_response('polls/index.html', {'latest_poll_list': latest_poll_list})
```

Note that once we've done this in all these views, we no longer need to import `loader`, `Context` and `HttpResponse`.

The `render_to_response()` function takes a template name as its first argument and a dictionary as its optional second argument. It returns an `HttpResponse` object of the given template rendered with the given context.

2.5.5 Raising 404

Now, let's tackle the poll detail view – the page that displays the question for a given poll. Here's the view:

```
from django.http import Http404
# ...
def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render_to_response('polls/detail.html', {'poll': p})
```

The new concept here: The view raises the `Http404` exception if a poll with the requested ID doesn't exist.

We'll discuss what you could put in that `polls/detail.html` template a bit later, but if you'd like to quickly get the above example working, just:

```
{{ poll }}
```

will get you started for now.

A shortcut: get_object_or_404()

It's a very common idiom to use `get()` and raise `Http404` if the object doesn't exist. Django provides a shortcut. Here's the `detail()` view, rewritten:

```
from django.shortcuts import render_to_response, get_object_or_404
# ...
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': p})
```

The `get_object_or_404()` function takes a Django model as its first argument and an arbitrary number of keyword arguments, which it passes to the `get()` function of the model's manager. It raises `Http404` if the object doesn't exist.

Philosophy

Why do we use a helper function `get_object_or_404()` instead of automatically catching the `ObjectDoesNotExist` exceptions at a higher level, or having the model API raise `Http404` instead of `ObjectDoesNotExist`?

Because that would couple the model layer to the view layer. One of the foremost design goals of Django is to maintain loose coupling.

There's also a `get_list_or_404()` function, which works just as `get_object_or_404()` – except using `filter()` instead of `get()`. It raises `Http404` if the list is empty.

2.5.6 Write a 404 (page not found) view

When you raise `Http404` from within a view, Django will load a special view devoted to handling 404 errors. It finds it by looking for the variable `handler404` in your root URLconf (and only in your root URLconf; setting `handler404` anywhere else will have no effect), which is a string in Python dotted syntax – the same format the normal URLconf callbacks use. A 404 view itself has nothing special: It's just a normal view.

You normally won't have to bother with writing 404 views. If you don't set `handler404`, the built-in view `django.views.defaults.page_not_found()` is used by default. In this case, you still have one obligation: create a `404.html` template in the root of your template directory. The default 404 view will use that template for all 404 errors. If `DEBUG` is set to `False` (in your settings module) and if you didn't create a `404.html` file, an `Http500` is raised instead. So remember to create a `404.html`.

A couple more things to note about 404 views:

- If `DEBUG` is set to `True` (in your settings module) then your 404 view will never be used (and thus the `404.html` template will never be rendered) because the traceback will be displayed instead.
- The 404 view is also called if Django doesn't find a match after checking every regular expression in the URLconf.

2.5.7 Write a 500 (server error) view

Similarly, your root URLconf may define a `handler500`, which points to a view to call in case of server errors. Server errors happen when you have runtime errors in view code.

2.5.8 Use the template system

Back to the `detail()` view for our poll application. Given the context variable `poll`, here's what the "polls/detail.html" template might look like:

```
<h1>{{ poll.question }}</h1>
<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }}</li>
{% endfor %}
</ul>
```

The template system uses dot-lookup syntax to access variable attributes. In the example of `{{ poll.question }}`, first Django does a dictionary lookup on the object `poll`. Failing that, it tries an attribute lookup – which works, in this case. If attribute lookup had failed, it would've tried a list-index lookup.

Method-calling happens in the `{% for %}` loop: `poll.choice_set.all` is interpreted as the Python code `poll.choice_set.all()`, which returns an iterable of `Choice` objects and is suitable for use in the `{% for %}` tag.

See the *template guide* for more about templates.

2.5.9 Simplifying the URLconfs

Take some time to play around with the views and template system. As you edit the URLconf, you may notice there's a fair bit of redundancy in it:

```
urlpatterns = patterns('',
    url(r'^polls/$', 'polls.views.index'),
    url(r'^polls/(?P<poll_id>\d+)/$', 'polls.views.detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'polls.views.results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'polls.views.vote'),
)
```

Namely, `polls.views` is in every callback.

Because this is a common case, the URLconf framework provides a shortcut for common prefixes. You can factor out the common prefixes and add them as the first argument to `patterns()`, like so:

```
urlpatterns = patterns('polls.views',
    url(r'^polls/$', 'index'),
    url(r'^polls/(?P<poll_id>\d+)/$', 'detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

This is functionally identical to the previous formatting. It's just a bit tidier.

Since you generally don't want the prefix for one app to be applied to every callback in your URLconf, you can concatenate multiple `patterns()`. Your full `mysite/urls.py` might now look like this:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('polls.views',
    url(r'^polls/$', 'index'),
    url(r'^polls/(?P<poll_id>\d+)/$', 'detail'),
    url(r'^polls/(?P<poll_id>\d+)/results/$', 'results'),
    url(r'^polls/(?P<poll_id>\d+)/vote/$', 'vote'),
)

urlpatterns += patterns('',
    url(r'^admin/', include(admin.site.urls)),
)
```

2.5.10 Decoupling the URLconfs

While we're at it, we should take the time to decouple our poll-app URLs from our Django project configuration. Django apps are meant to be pluggable – that is, each particular app should be transferable to another Django installation with minimal fuss.

Our poll app is pretty decoupled at this point, thanks to the strict directory structure that `python manage.py startapp` created, but one part of it is coupled to the Django settings: The URLconf.

We've been editing the URLs in `mysite/urls.py`, but the URL design of an app is specific to the app, not to the Django installation – so let's move the URLs within the app directory.

Copy the file `mysite/urls.py` to `polls/urls.py`. Then, change `mysite/urls.py` to remove the poll-specific URLs and insert an `include()`, leaving you with:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^polls/', include('polls.urls')),
    url(r'^admin/', include(admin.site.urls)),
)
```

`include()` simply references another URLconf. Note that the regular expression doesn't have a `$` (end-of-string match character) but has the trailing slash. Whenever Django encounters `include()`, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Here's what happens if a user goes to "/polls/34/" in this system:

- Django will find the match at `'^polls/'`
- Then, Django will strip off the matching text (`"polls/"`) and send the remaining text – `"34/"` – to the 'polls.urls' URLconf for further processing.

Now that we've decoupled that, we need to decouple the `polls.urls` URLconf by removing the leading "polls/" from each line, and removing the lines registering the admin site. Your `polls/urls.py` file should now look like this:

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('polls.views',
    url(r'^$', 'index'),
    url(r'^(?P<poll_id>\d+)/$', 'detail'),
    url(r'^(?P<poll_id>\d+)/results/$', 'results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

The idea behind `include()` and URLconf decoupling is to make it easy to plug-and-play URLs. Now that polls are in their own URLconf, they can be placed under "/polls/", or under "/fun_polls/", or under "/content/polls/", or any other path root, and the app will still work.

All the poll app cares about is its relative path, not its absolute path.

When you're comfortable with writing views, read *part 4 of this tutorial* to learn about simple form processing and generic views.

2.6 Writing your first Django app, part 4

This tutorial begins where *Tutorial 3* left off. We're continuing the Web-poll application and will focus on simple form processing and cutting down our code.

2.6.1 Write a simple form

Let's update our poll detail template ("polls/detail.html") from the last tutorial, so that the template contains an HTML `<form>` element:

```
<h1>{{ poll.question }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="/polls/{{ poll.id }}/vote/" method="post">
{% csrf_token %}
{% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```

A quick rundown:

- The above template displays a radio button for each poll choice. The `value` of each radio button is the associated poll choice's ID. The `name` of each radio button is `"choice"`. That means, when somebody selects one of the radio buttons and submits the form, it'll send the POST data `choice=3`. This is HTML Forms 101.
- We set the form's `action` to `/polls/{{ poll.id }}/vote/`, and we set `method="post"`. Using `method="post"` (as opposed to `method="get"`) is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use `method="post"`. This tip isn't specific to Django; it's just good Web development practice.
- `forloop.counter` indicates how many times the `for` tag has gone through its loop
- Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a very easy-to-use system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the `{% csrf_token %}` template tag.

The `{% csrf_token %}` tag requires information from the request object, which is not normally accessible from within the template context. To fix this, a small adjustment needs to be made to the `detail` view, so that it looks like the following:

```
from django.template import RequestContext
# ...
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': p},
                              context_instance=RequestContext(request))
```

The details of how this works are explained in the documentation for *RequestContext*.

Now, let's create a Django view that handles the submitted data and does something with it. Remember, in *Tutorial 3*, we created a URLconf for the polls application that includes this line:

```
(r'^(?P<poll_id>\d+)/vote/$', 'vote'),
```

We also created a dummy implementation of the `vote()` function. Let's create a real version. Add the following to `polls/views.py`:

```
from django.shortcuts import get_object_or_404, render_to_response
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
from django.template import RequestContext
from polls.models import Choice, Poll
# ...
def vote(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
```

```
    try:
        selected_choice = p.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the poll voting form.
        return render_to_response('polls/detail.html', {
            'poll': p,
            'error_message': "You didn't select a choice.",
        }, context_instance=RequestContext(request))
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.
        return HttpResponseRedirect(reverse('polls.views.results', args=(p.id,)))
```

This code includes a few things we haven't covered yet in this tutorial:

- `request.POST` is a dictionary-like object that lets you access submitted data by key name. In this case, `request.POST['choice']` returns the ID of the selected choice, as a string. `request.POST` values are always strings.

  Note that Django also provides `request.GET` for accessing GET data in the same way – but we're explicitly using `request.POST` in our code, to ensure that data is only altered via a POST call.

- `request.POST['choice']` will raise `KeyError` if `choice` wasn't provided in POST data. The above code checks for `KeyError` and redisplays the poll form with an error message if `choice` isn't given.

- After incrementing the choice count, the code returns an `HttpResponseRedirect` rather than a normal `HttpResponse`. `HttpResponseRedirect` takes a single argument: the URL to which the user will be redirected (see the following point for how we construct the URL in this case).

  As the Python comment above points out, you should always return an `HttpResponseRedirect` after successfully dealing with POST data. This tip isn't specific to Django; it's just good Web development practice.

- We are using the `reverse()` function in the `HttpResponseRedirect` constructor in this example. This function helps avoid having to hardcode a URL in the view function. It is given the name of the view that we want to pass control to and the variable portion of the URL pattern that points to that view. In this case, using the URLconf we set up in Tutorial 3, this `reverse()` call will return a string like

  ```
  '/polls/3/results/'
  ```

  ... where the `3` is the value of `p.id`. This redirected URL will then call the `'results'` view to display the final page. Note that you need to use the full name of the view here (including the prefix).

As mentioned in Tutorial 3, `request` is a `HttpRequest` object. For more on `HttpRequest` objects, see the *request and response documentation*.

After somebody votes in a poll, the `vote()` view redirects to the results page for the poll. Let's write that view:

```
def results(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/results.html', {'poll': p})
```

This is almost exactly the same as the `detail()` view from *Tutorial 3*. The only difference is the template name. We'll fix this redundancy later.

Now, create a `results.html` template:

```
<h1>{{ poll.question }}</h1>
```

```
<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>

<a href="/polls/{{ poll.id }}/">Vote again?</a>
```

Now, go to `/polls/1/` in your browser and vote in the poll. You should see a results page that gets updated each time you vote. If you submit the form without having chosen a choice, you should see the error message.

2.6.2 Use generic views: Less code is better

The `detail()` (from *Tutorial 3*) and `results()` views are stupidly simple – and, as mentioned above, redundant. The `index()` view (also from Tutorial 3), which displays a list of polls, is similar.

These views represent a common case of basic Web development: getting data from the database according to a parameter passed in the URL, loading a template and returning the rendered template. Because this is so common, Django provides a shortcut, called the "generic views" system.

Generic views abstract common patterns to the point where you don't even need to write Python code to write an app.

Let's convert our poll app to use the generic views system, so we can delete a bunch of our own code. We'll just have to take a few steps to make the conversion. We will:

1. Convert the URLconf.
2. Delete some of the old, unneeded views.
3. Fix up URL handling for the new views.

Read on for details.

Why the code-shuffle?

Generally, when writing a Django app, you'll evaluate whether generic views are a good fit for your problem, and you'll use them from the beginning, rather than refactoring your code halfway through. But this tutorial intentionally has focused on writing the views "the hard way" until now, to focus on core concepts.

You should know basic math before you start using a calculator.

First, open the `polls/urls.py` URLconf. It looks like this, according to the tutorial so far:

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('polls.views',
    url(r'^$', 'index'),
    url(r'^(?P<poll_id>\d+)/$', 'detail'),
    url(r'^(?P<poll_id>\d+)/results/$', 'results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

Change it like so:

```
from django.conf.urls import patterns, include, url
from django.views.generic import DetailView, ListView
from polls.models import Poll

urlpatterns = patterns('',
```

```
    url(r'^$',
        ListView.as_view(
            queryset=Poll.objects.order_by('-pub_date')[:5],
            context_object_name='latest_poll_list',
            template_name='polls/index.html')),
    url(r'^(?P<pk>\d+)/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/detail.html')),
    url(r'^(?P<pk>\d+)/results/$',
        DetailView.as_view(
            model=Poll,
            template_name='polls/results.html'),
        name='poll_results'),
    url(r'^(?P<poll_id>\d+)/vote/$', 'polls.views.vote'),
)
```

We're using two generic views here: `ListView` and `DetailView`. Respectively, those two views abstract the concepts of "display a list of objects" and "display a detail page for a particular type of object."

- Each generic view needs to know what model it will be acting upon. This is provided using the `model` parameter.
- The `DetailView` generic view expects the primary key value captured from the URL to be called `"pk"`, so we've changed `poll_id` to `pk` for the generic views.
- We've added a name, `poll_results`, to the results view so that we have a way to refer to its URL later on (see the documentation about *naming URL patterns* for information). We're also using the `url()` function from `django.conf.urls` here. It's a good habit to use `url()` when you are providing a pattern name like this.

By default, the `DetailView` generic view uses a template called `<app name>/<model name>_detail.html`. In our case, it'll use the template `"polls/poll_detail.html"`. The `template_name` argument is used to tell Django to use a specific template name instead of the autogenerated default template name. We also specify the `template_name` for the `results` list view – this ensures that the results view and the detail view have a different appearance when rendered, even though they're both a `DetailView` behind the scenes.

Similarly, the `ListView` generic view uses a default template called `<app name>/<model name>_list.html`; we use `template_name` to tell `ListView` to use our existing `"polls/index.html"` template.

In previous parts of the tutorial, the templates have been provided with a context that contains the `poll` and `latest_poll_list` context variables. For DetailView the `poll` variable is provided automatically – since we're using a Django model (`Poll`), Django is able to determine an appropriate name for the context variable. However, for ListView, the automatically generated context variable is `poll_list`. To override this we provide the `context_object_name` option, specifying that we want to use `latest_poll_list` instead. As an alternative approach, you could change your templates to match the new default context variables – but it's a lot easier to just tell Django to use the variable you want.

You can now delete the `index()`, `detail()` and `results()` views from `polls/views.py`. We don't need them anymore – they have been replaced by generic views.

The last thing to do is fix the URL handling to account for the use of generic views. In the vote view above, we used the `reverse()` function to avoid hard-coding our URLs. Now that we've switched to a generic view, we'll need to change the `reverse()` call to point back to our new generic view. We can't simply use the view function anymore – generic views can be (and are) used multiple times – but we can use the name we've given:

```
return HttpResponseRedirect(reverse('poll_results', args=(p.id,)))
```

Run the server, and use your new polling app based on generic views.

For full details on generic views, see the *generic views documentation*.

2.6.3 Coming soon

The tutorial ends here for the time being. Future installments of the tutorial will cover:

- Advanced form processing
- Using the RSS framework
- Using the cache framework
- Using the comments framework
- Advanced admin features: Permissions
- Advanced admin features: Custom JavaScript

In the meantime, you might want to check out some pointers on *where to go from here*

2.7 What to read next

So you've read all the *introductory material* and have decided you'd like to keep using Django. We've only just scratched the surface with this intro (in fact, if you've read every single word you've still read less than 10% of the overall documentation).

So what's next?

Well, we've always been big fans of learning by doing. At this point you should know enough to start a project of your own and start fooling around. As you need to learn new tricks, come back to the documentation.

We've put a lot of effort into making Django's documentation useful, easy to read and as complete as possible. The rest of this document explains more about how the documentation works so that you can get the most out of it.

(Yes, this is documentation about documentation. Rest assured we have no plans to write a document about how to read the document about documentation.)

2.7.1 Finding documentation

Django's got a *lot* of documentation – almost 200,000 words – so finding what you need can sometimes be tricky. A few good places to start are the *search* and the *genindex*.

Or you can just browse around!

2.7.2 How the documentation is organized

Django's main documentation is broken up into "chunks" designed to fill different needs:

- The *introductory material* is designed for people new to Django – or to Web development in general. It doesn't cover anything in depth, but instead gives a high-level overview of how developing in Django "feels".

- The *topic guides*, on the other hand, dive deep into individual parts of Django. There are complete guides to Django's *model system*, *template engine*, *forms framework*, and much more.

  This is probably where you'll want to spend most of your time; if you work your way through these guides you should come out knowing pretty much everything there is to know about Django.

- Web development is often broad, not deep – problems span many domains. We've written a set of *how-to guides* that answer common "How do I ...?" questions. Here you'll find information about *generating PDFs with Django*, *writing custom template tags*, and more.

  Answers to really common questions can also be found in the *FAQ*.

- The guides and how-to's don't cover every single class, function, and method available in Django – that would be overwhelming when you're trying to learn. Instead, details about individual classes, functions, methods, and modules are kept in the *reference*. This is where you'll turn to find the details of a particular function or whathaveyou.

- Finally, there's some "specialized" documentation not usually relevant to most developers. This includes the *release notes*, *documentation of obsolete features*, *internals documentation* for those who want to add code to Django itself, and a *few other things that simply don't fit elsewhere*.

2.7.3 How documentation is updated

Just as the Django code base is developed and improved on a daily basis, our documentation is consistently improving. We improve documentation for several reasons:

- To make content fixes, such as grammar/typo corrections.
- To add information and/or examples to existing sections that need to be expanded.
- To document Django features that aren't yet documented. (The list of such features is shrinking but exists nonetheless.)
- To add documentation for new features as new features get added, or as Django APIs or behaviors change.

Django's documentation is kept in the same source control system as its code. It lives in the docs directory of our Git repository. Each document online is a separate text file in the repository.

2.7.4 Where to get it

You can read Django documentation in several ways. They are, in order of preference:

On the Web

The most recent version of the Django documentation lives at https://docs.djangoproject.com/en/dev/. These HTML pages are generated automatically from the text files in source control. That means they reflect the "latest and greatest" in Django – they include the very latest corrections and additions, and they discuss the latest Django features, which may only be available to users of the Django development version. (See "Differences between versions" below.)

We encourage you to help improve the docs by submitting changes, corrections and suggestions in the ticket system. The Django developers actively monitor the ticket system and use your feedback to improve the documentation for everybody.

Note, however, that tickets should explicitly relate to the documentation, rather than asking broad tech-support questions. If you need help with your particular Django setup, try the django-users mailing list or the #django IRC channel instead.

In plain text

For offline reading, or just for convenience, you can read the Django documentation in plain text.

If you're using an official release of Django, note that the zipped package (tarball) of the code includes a `docs/` directory, which contains all the documentation for that release.

If you're using the development version of Django (aka "trunk"), note that the `docs/` directory contains all of the documentation. You can update your Git checkout to get the latest changes.

One low-tech way of taking advantage of the text documentation is by using the Unix `grep` utility to search for a phrase in all of the documentation. For example, this will show you each mention of the phrase "max_length" in any Django document:

```
$ grep -r max_length /path/to/django/docs/
```

As HTML, locally

You can get a local copy of the HTML documentation following a few easy steps:

- Django's documentation uses a system called Sphinx to convert from plain text to HTML. You'll need to install Sphinx by either downloading and installing the package from the Sphinx Web site, or with `pip`:

  ```
  $ sudo pip install Sphinx
  ```

- Then, just use the included `Makefile` to turn the documentation into HTML:

  ```
  $ cd path/to/django/docs
  $ make html
  ```

  You'll need GNU Make installed for this.

  If you're on Windows you can alternatively use the included batch file:

  ```
  cd path\to\django\docs
  make.bat html
  ```

- The HTML documentation will be placed in `docs/_build/html`.

Note: Generation of the Django documentation will work with Sphinx version 0.6 or newer, but we recommend going straight to Sphinx 1.0.2 or newer.

2.7.5 Differences between versions

As previously mentioned, the text documentation in our Git repository contains the "latest and greatest" changes and additions. These changes often include documentation of new features added in the Django development version – the Git ("trunk") version of Django. For that reason, it's worth pointing out our policy on keeping straight the documentation for various versions of the framework.

We follow this policy:

- The primary documentation on djangoproject.com is an HTML version of the latest docs in Git. These docs always correspond to the latest official Django release, plus whatever features we've added/changed in the framework *since* the latest release.
- As we add features to Django's development version, we try to update the documentation in the same Git commit transaction.

- To distinguish feature changes/additions in the docs, we use the phrase: "New in version X.Y", being X.Y the next release version (hence, the one being developed).
- Documentation for a particular Django release is frozen once the version has been released officially. It remains a snapshot of the docs as of the moment of the release. We will make exceptions to this rule in the case of retroactive security updates or other such retroactive changes. Once documentation is frozen, we add a note to the top of each frozen document that says "These docs are frozen for Django version XXX" and links to the current version of that document.
- The main documentation Web page includes links to documentation for all previous versions.

See Also:

If you're new to Python, you might want to start by getting an idea of what the language is like. Django is 100% Python, so if you've got minimal comfort with Python you'll probably get a lot more out of Django.

If you're new to programming entirely, you might want to start with this list of Python resources for non-programmers

If you already know a few other languages and want to get up to speed with Python quickly, we recommend Dive Into Python (also available in a dead-tree version). If that's not quite your style, there are quite a few other books about Python.

CHAPTER

THREE

USING DJANGO

Introductions to all the key parts of Django you'll need to know:

3.1 How to install Django

This document will get you up and running with Django.

3.1.1 Install Python

Being a Python Web framework, Django requires Python.

It works with any Python version from 2.6.5 to 2.7 (due to backwards incompatibilities in Python 3.0, Django does not currently work with Python 3.0; see *the Django FAQ* for more information on supported Python versions and the 3.0 transition).

Get Python at http://www.python.org. If you're running Linux or Mac OS X, you probably already have it installed.

Django on Jython

If you use Jython (a Python implementation for the Java platform), you'll need to follow a few additional steps. See *Running Django on Jython* for details.

Python on Windows

On Windows, you might need to adjust your `PATH` environment variable to include paths to Python executable and additional scripts. For example, if your Python is installed in `C:\Python27\`, the following paths need to be added to `PATH`:

```
C:\Python27\;C:\Python27\Scripts;
```

3.1.2 Install Apache and mod_wsgi

If you just want to experiment with Django, skip ahead to the next section; Django includes a lightweight web server you can use for testing, so you won't need to set up Apache until you're ready to deploy Django in production.

If you want to use Django on a production site, use Apache with mod_wsgi. mod_wsgi can operate in one of two modes: an embedded mode and a daemon mode. In embedded mode, mod_wsgi is similar to mod_perl – it embeds Python within Apache and loads Python code into memory when the server starts. Code stays in memory throughout

the life of an Apache process, which leads to significant performance gains over other server arrangements. In daemon mode, mod_wsgi spawns an independent daemon process that handles requests. The daemon process can run as a different user than the Web server, possibly leading to improved security, and the daemon process can be restarted without restarting the entire Apache Web server, possibly making refreshing your codebase more seamless. Consult the mod_wsgi documentation to determine which mode is right for your setup. Make sure you have Apache installed, with the mod_wsgi module activated. Django will work with any version of Apache that supports mod_wsgi.

See *How to use Django with mod_wsgi* for information on how to configure mod_wsgi once you have it installed.

If you can't use mod_wsgi for some reason, fear not: Django supports many other deployment options. One is *uWSGI*; it works very well with nginx. Another is *FastCGI*, perfect for using Django with servers other than Apache. Additionally, Django follows the WSGI spec (**PEP 3333**), which allows it to run on a variety of server platforms. See the server-arrangements wiki page for specific installation instructions for each platform.

3.1.3 Get your database running

If you plan to use Django's database API functionality, you'll need to make sure a database server is running. Django supports many different database servers and is officially supported with PostgreSQL, MySQL, Oracle and SQLite (although SQLite doesn't require a separate server to be running).

In addition to the officially supported databases, there are backends provided by 3rd parties that allow you to use other databases with Django:

- Sybase SQL Anywhere
- IBM DB2
- Microsoft SQL Server 2005
- Firebird
- ODBC

The Django versions and ORM features supported by these unofficial backends vary considerably. Queries regarding the specific capabilities of these unofficial backends, along with any support queries, should be directed to the support channels provided by each 3rd party project.

In addition to a database backend, you'll need to make sure your Python database bindings are installed.

- If you're using PostgreSQL, you'll need the `postgresql_psycopg2` package. You might want to refer to our *PostgreSQL notes* for further technical details specific to this database.

  If you're on Windows, check out the unofficial compiled Windows version.

- If you're using MySQL, you'll need the `MySQL-python` package, version 1.2.1p2 or higher. You will also want to read the database-specific *notes for the MySQL backend*.

- If you're using Oracle, you'll need a copy of cx_Oracle, but please read the database-specific *notes for the Oracle backend* for important information regarding supported versions of both Oracle and `cx_Oracle`.

- If you're using an unofficial 3rd party backend, please consult the documentation provided for any additional requirements.

If you plan to use Django's `manage.py syncdb` command to automatically create database tables for your models, you'll need to ensure that Django has permission to create and alter tables in the database you're using; if you plan to manually create the tables, you can simply grant Django `SELECT`, `INSERT`, `UPDATE` and `DELETE` permissions. On some databases, Django will need `ALTER TABLE` privileges during `syncdb` but won't issue `ALTER TABLE` statements on a table once `syncdb` has created it.

If you're using Django's *testing framework* to test database queries, Django will need permission to create a test database.

3.1.4 Remove any old versions of Django

If you are upgrading your installation of Django from a previous version, you will need to uninstall the old Django version before installing the new version.

If you installed Django using pip or `easy_install` previously, installing with pip or `easy_install` again will automatically take care of the old version, so you don't need to do it yourself.

If you previously installed Django using `python setup.py install`, uninstalling is as simple as deleting the `django` directory from your Python `site-packages`. To find the directory you need to remove, you can run the following at your shell prompt (not the interactive Python prompt):

```
python -c "import sys; sys.path = sys.path[1:]; import django; print(django.__path__)"
```

3.1.5 Install the Django code

Installation instructions are slightly different depending on whether you're installing a distribution-specific package, downloading the latest official release, or fetching the latest development version.

It's easy, no matter which way you choose.

Installing a distribution-specific package

Check the *distribution specific notes* to see if your platform/distribution provides official Django packages/installers. Distribution-provided packages will typically allow for automatic installation of dependencies and easy upgrade paths.

Installing an official release with `pip`

This is the recommended way to install Django.

1. Install pip. The easiest is to use the standalone pip installer. If your distribution already has `pip` installed, you might need to update it if it's outdated. (If it's outdated, you'll know because installation won't work.)
2. (optional) Take a look at virtualenv and virtualenvwrapper. These tools provide isolated Python environments, which are more practical than installing packages systemwide. They also allow installing packages without administrator privileges. It's up to you to decide if you want to learn and use them.
3. If you're using Linux, Mac OS X or some other flavor of Unix, enter the command `sudo pip install Django` at the shell prompt. If you're using Windows, start a command shell with administrator privileges and run the command `pip install Django`. This will install Django in your Python installation's `site-packages` directory.

   If you're using a virtualenv, you don't need `sudo` or administrator privileges, and this will install Django in the virtualenv's `site-packages` directory.

Installing an official release manually

1. Download the latest release from our download page.
2. Untar the downloaded file (e.g. `tar xzvf Django-X.Y.tar.gz`, where `X.Y` is the version number of the latest release). If you're using Windows, you can download the command-line tool bsdtar to do this, or you can use a GUI-based tool such as 7-zip.
3. Change into the directory created in step 2 (e.g. `cd Django-X.Y`).

4. If you're using Linux, Mac OS X or some other flavor of Unix, enter the command `sudo python setup.py install` at the shell prompt. If you're using Windows, start a command shell with administrator privileges and run the command `python setup.py install`. This will install Django in your Python installation's `site-packages` directory.

Removing an old version

If you use this installation technique, it is particularly important that you *remove any existing installations* of Django first. Otherwise, you can end up with a broken installation that includes files from previous versions that have since been removed from Django.

Installing the development version

Tracking Django development

If you decide to use the latest development version of Django, you'll want to pay close attention to the development timeline, and you'll want to keep an eye on the list of backwards-incompatible changes. This will help you stay on top of any new features you might want to use, as well as any changes you'll need to make to your code when updating your copy of Django. (For stable releases, any necessary changes are documented in the release notes.)

If you'd like to be able to update your Django code occasionally with the latest bug fixes and improvements, follow these instructions:

1. Make sure that you have Git installed and that you can run its commands from a shell. (Enter `git help` at a shell prompt to test this.)
2. Check out Django's main development branch (the 'trunk' or 'master') like so:

   ```
   git clone git://github.com/django/django.git django-trunk
   ```

   This will create a directory `django-trunk` in your current directory.
3. Make sure that the Python interpreter can load Django's code. The most convenient way to do this is via pip. Run the following command:

   ```
   sudo pip install -e django-trunk/
   ```

   (If using a virtualenv you can omit `sudo`.)

   This will make Django's code importable, and will also make the `django-admin.py` utility command available. In other words, you're all set!

   If you don't have pip available, see the alternative instructions for installing the development version without pip.

Warning: Don't run `sudo python setup.py install`, because you've already carried out the equivalent actions in step 3.

When you want to update your copy of the Django source code, just run the command `git pull` from within the `django-trunk` directory. When you do this, Git will automatically download any changes.

Installing the development version without pip

If you don't have pip, you can instead manually modify Python's search path.

First follow steps 1 and 2 above, so that you have a `django-trunk` directory with a checkout of Django's latest code in it. Then add a `.pth` file containing the full path to the `django-trunk` directory to your system's `site-packages` directory. For example, on a Unix-like system:

```
echo WORKING-DIR/django-trunk > SITE-PACKAGES-DIR/django.pth
```

In the above line, change `WORKING-DIR/django-trunk` to match the full path to your new `django-trunk` directory, and change `SITE-PACKAGES-DIR` to match the location of your system's `site-packages` directory.

The location of the `site-packages` directory depends on the operating system, and the location in which Python was installed. To find your system's `site-packages` location, execute the following:

```
python -c "from distutils.sysconfig import get_python_lib; print(get_python_lib())"
```

(Note that this should be run from a shell prompt, not a Python interactive prompt.)

Some Debian-based Linux distributions have separate `site-packages` directories for user-installed packages, such as when installing Django from a downloaded tarball. The command listed above will give you the system's `site-packages`, the user's directory can be found in `/usr/local/lib/` instead of `/usr/lib/`.

Next you need to make the `django-admin.py` utility available in your shell PATH.

On Unix-like systems, create a symbolic link to the file `django-trunk/django/bin/django-admin.py` in a directory on your system path, such as `/usr/local/bin`. For example:

```
ln -s WORKING-DIR/django-trunk/django/bin/django-admin.py /usr/local/bin/
```

(In the above line, change WORKING-DIR to match the full path to your new `django-trunk` directory.)

This simply lets you type `django-admin.py` from within any directory, rather than having to qualify the command with the full path to the file.

On Windows systems, the same result can be achieved by copying the file `django-trunk/django/bin/django-admin.py` to somewhere on your system path, for example `C:\Python27\Scripts`.

3.2 Models and databases

A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

3.2.1 Models

A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses `django.db.models.Model`.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API; see *Making queries*.

Quick example

This example model defines a `Person`, which has a `first_name` and `last_name`:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` and `last_name` are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above `Person` model would create a database table like this:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Some technical notes:

- The name of the table, `myapp_person`, is automatically derived from some model metadata but can be overridden. See *Table names* for more details..
- An `id` field is added automatically, but this behavior can be overridden. See *Automatic primary key fields*.
- The `CREATE TABLE` SQL in this example is formatted using PostgreSQL syntax, but it's worth noting Django uses SQL tailored to the database backend specified in your *settings file*.

Using models

Once you have defined your models, you need to tell Django you're going to *use* those models. Do this by editing your settings file and changing the `INSTALLED_APPS` setting to add the name of the module that contains your `models.py`.

For example, if the models for your application live in the module `mysite.myapp.models` (the package structure that is created for an application by the `manage.py startapp` script), `INSTALLED_APPS` should read, in part:

```
INSTALLED_APPS = (
    #...
    'mysite.myapp',
    #...
)
```

When you add new apps to `INSTALLED_APPS`, be sure to run `manage.py syncdb`.

Fields

The most important part of a model – and the only required part of a model – is the list of database fields it defines. Fields are specified by class attributes.

Example:

```
class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
```

```
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Field types

Each field in your model should be an instance of the appropriate `Field` class. Django uses the field class types to determine a few things:

- The database column type (e.g. `INTEGER`, `VARCHAR`).
- The *widget* to use in Django's admin interface, if you care to use it (e.g. `<input type="text">`, `<select>`).
- The minimal validation requirements, used in Django's admin and in automatically-generated forms.

Django ships with dozens of built-in field types; you can find the complete list in the *model field reference*. You can easily write your own fields if Django's built-in ones don't do the trick; see *Writing custom model fields*.

Field options

Each field takes a certain set of field-specific arguments (documented in the *model field reference*). For example, `CharField` (and its subclasses) require a `max_length` argument which specifies the size of the `VARCHAR` database field used to store the data.

There's also a set of common arguments available to all field types. All are optional. They're fully explained in the *reference*, but here's a quick summary of the most often-used ones:

`null` If `True`, Django will store empty values as `NULL` in the database. Default is `False`.

`blank` If `True`, the field is allowed to be blank. Default is `False`.

Note that this is different than `null`. `null` is purely database-related, whereas `blank` is validation-related. If a field has `blank=True`, validation on Django's admin site will allow entry of an empty value. If a field has `blank=False`, the field will be required.

`choices` An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field. If this is given, Django's admin will use a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

The first element in each tuple is the value that will be stored in the database, the second element will be displayed by the admin interface, or in a ModelChoiceField. Given an instance of a model object, the display value for a choices field can be accessed using the `get_FOO_display` method. For example:

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
```

```
            ('M', 'Medium'),
            ('L', 'Large'),
        )
        name = models.CharField(max_length=60)
        shirt_size = models.CharField(max_length=2, choices=SHIRT_SIZES)

>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
u'L'
>>> p.get_shirt_size_display()
u'Large'
```

default The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

help_text Extra "help" text to be displayed under the field on the object's admin form. It's useful for documentation even if your object doesn't have an admin form.

primary_key If `True`, this field is the primary key for the model.

If you don't specify `primary_key=True` for any fields in your model, Django will automatically add an `IntegerField` to hold the primary key, so you don't need to set `primary_key=True` on any of your fields unless you want to override the default primary-key behavior. For more, see *Automatic primary key fields*.

unique If `True`, this field must be unique throughout the table.

Again, these are just short descriptions of the most common field options. Full details can be found in the *common model field option reference*.

Automatic primary key fields

By default, Django gives each model the following field:

```
id = models.AutoField(primary_key=True)
```

This is an auto-incrementing primary key.

If you'd like to specify a custom primary key, just specify `primary_key=True` on one of your fields. If Django sees you've explicitly set `Field.primary_key`, it won't add the automatic `id` column.

Each model requires exactly one field to have `primary_key=True`.

Verbose field names

Each field type, except for `ForeignKey`, `ManyToManyField` and `OneToOneField`, takes an optional first positional argument – a verbose name. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.

In this example, the verbose name is `"person's first name"`:

```
first_name = models.CharField("person's first name", max_length=30)
```

In this example, the verbose name is `"first name"`:

```
first_name = models.CharField(max_length=30)
```

`ForeignKey`, `ManyToManyField` and `OneToOneField` require the first argument to be a model class, so use the `verbose_name` keyword argument:

```
poll = models.ForeignKey(Poll, verbose_name="the related poll")
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(Place, verbose_name="related place")
```

The convention is not to capitalize the first letter of the `verbose_name`. Django will automatically capitalize the first letter where it needs to.

Relationships

Clearly, the power of relational databases lies in relating tables to each other. Django offers ways to define the three most common types of database relationships: many-to-one, many-to-many and one-to-one.

Many-to-one relationships To define a many-to-one relationship, use `django.db.models.ForeignKey`. You use it just like any other `Field` type: by including it as a class attribute of your model.

`ForeignKey` requires a positional argument: the class to which the model is related.

For example, if a `Car` model has a `Manufacturer` – that is, a `Manufacturer` makes multiple cars but each `Car` only has one `Manufacturer` – use the following definitions:

```
class Manufacturer(models.Model):
    # ...

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer)
    # ...
```

You can also create *recursive relationships* (an object with a many-to-one relationship to itself) and *relationships to models not yet defined*; see *the model field reference* for details.

It's suggested, but not required, that the name of a `ForeignKey` field (`manufacturer` in the example above) be the name of the model, lowercase. You can, of course, call the field whatever you want. For example:

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(Manufacturer)
    # ...
```

See Also:

`ForeignKey` fields accept a number of extra arguments which are explained in *the model field reference*. These options help define how the relationship should work; all are optional.

For details on accessing backwards-related objects, see the *Following relationships backward example*.

For sample code, see the *Many-to-one relationship model example*.

Many-to-many relationships To define a many-to-many relationship, use `ManyToManyField`. You use it just like any other `Field` type: by including it as a class attribute of your model.

`ManyToManyField` requires a positional argument: the class to which the model is related.

For example, if a `Pizza` has multiple `Topping` objects – that is, a `Topping` can be on multiple pizzas and each `Pizza` has multiple toppings – here's how you'd represent that:

```
class Topping(models.Model):
    # ...

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

As with `ForeignKey`, you can also create *recursive relationships* (an object with a many-to-many relationship to itself) and *relationships to models not yet defined*; see *the model field reference* for details.

It's suggested, but not required, that the name of a `ManyToManyField` (`toppings` in the example above) be a plural describing the set of related model objects.

It doesn't matter which model has the `ManyToManyField`, but you should only put it in one of the models – not both.

Generally, `ManyToManyField` instances should go in the object that's going to be edited in the admin interface, if you're using Django's admin. In the above example, `toppings` is in `Pizza` (rather than `Topping` having a `pizzas` `ManyToManyField`) because it's more natural to think about a pizza having toppings than a topping being on multiple pizzas. The way it's set up above, the `Pizza` admin form would let users select the toppings.

See Also:

See the *Many-to-many relationship model example* for a full example.

`ManyToManyField` fields also accept a number of extra arguments which are explained in *the model field reference*. These options help define how the relationship should work; all are optional.

Extra fields on many-to-many relationships When you're only dealing with simple many-to-many relationships such as mixing and matching pizzas and toppings, a standard `ManyToManyField` is all you need. However, sometimes you may need to associate data with the relationship between two models.

For example, consider the case of an application tracking the musical groups which musicians belong to. There is a many-to-many relationship between a person and the groups of which they are a member, so you could use a `ManyToManyField` to represent this relationship. However, there is a lot of detail about the membership that you might want to collect, such as the date at which the person joined the group.

For these situations, Django allows you to specify the model that will be used to govern the many-to-many relationship. You can then put extra fields on the intermediate model. The intermediate model is associated with the `ManyToManyField` using the `through` argument to point to the model that will act as an intermediary. For our musician example, the code would look something like this:

```
class Person(models.Model):
    name = models.CharField(max_length=128)

    def __unicode__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __unicode__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
```

```
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

When you set up the intermediary model, you explicitly specify foreign keys to the models that are involved in the ManyToMany relation. This explicit declaration defines how the two models are related.

There are a few restrictions on the intermediate model:

- Your intermediate model must contain one - and *only* one - foreign key to the target model (this would be `Person` in our example). If you have more than one foreign key, a validation error will be raised.
- Your intermediate model must contain one - and *only* one - foreign key to the source model (this would be `Group` in our example). If you have more than one foreign key, a validation error will be raised.
- The only exception to this is a model which has a many-to-many relationship to itself, through an intermediary model. In this case, two foreign keys to the same model are permitted, but they will be treated as the two (different) sides of the many-to-many relation.
- When defining a many-to-many relationship from a model to itself, using an intermediary model, you *must* use `symmetrical=False` (see *the model field reference*).

Now that you have set up your `ManyToManyField` to use your intermediary model (`Membership`, in this case), you're ready to start creating some many-to-many relationships. You do this by creating instances of the intermediate model:

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason= "Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
[<Person: Ringo Starr>]
>>> ringo.group_set.all()
[<Group: The Beatles>]
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason= "Wanted to form a band.")
>>> beatles.members.all()
[<Person: Ringo Starr>, <Person: Paul McCartney>]
```

Unlike normal many-to-many fields, you *can't* use `add`, `create`, or assignment (i.e., `beatles.members = [...]`) to create relationships:

```
# THIS WILL NOT WORK
>>> beatles.members.add(john)
# NEITHER WILL THIS
>>> beatles.members.create(name="George Harrison")
# AND NEITHER WILL THIS
>>> beatles.members = [john, paul, ringo, george]
```

Why? You can't just create a relationship between a `Person` and a `Group` - you need to specify all the detail for the relationship required by the `Membership` model. The simple `add`, `create` and assignment calls don't provide a way to specify this extra detail. As a result, they are disabled for many-to-many relationships that use an intermediate model. The only way to create this type of relationship is to create instances of the intermediate model.

The `remove()` method is disabled for similar reasons. However, the `clear()` method can be used to remove all many-to-many relationships for an instance:

```
# Beatles have broken up
>>> beatles.members.clear()
```

Once you have established the many-to-many relationships by creating instances of your intermediate model, you can issue queries. Just as with normal many-to-many relationships, you can query using the attributes of the many-to-many-related model:

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
[<Group: The Beatles>]
```

As you are using an intermediate model, you can also query on its attributes:

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
[<Person: Ringo Starr]
```

If you need to access a membership's information you may do so by directly querying the `Membership` model:

```
>>> ringos_membership = Membership.objects.get(group=beatles, person=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
u'Needed a new drummer.'
```

Another way to access the same information is by querying the *many-to-many reverse relationship* from a `Person` object:

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
u'Needed a new drummer.'
```

One-to-one relationships To define a one-to-one relationship, use `OneToOneField`. You use it just like any other `Field` type: by including it as a class attribute of your model.

This is most useful on the primary key of an object when that object "extends" another object in some way.

`OneToOneField` requires a positional argument: the class to which the model is related.

For example, if you were building a database of "places", you would build pretty standard stuff such as address, phone number, etc. in the database. Then, if you wanted to build a database of restaurants on top of the places, instead of repeating yourself and replicating those fields in the `Restaurant` model, you could make `Restaurant` have a `OneToOneField` to `Place` (because a restaurant "is a" place; in fact, to handle this you'd typically use *inheritance*, which involves an implicit one-to-one relation).

As with `ForeignKey`, a *recursive relationship* can be defined and *references to as-yet undefined models* can be made; see *the model field reference* for details.

See Also:

See the *One-to-one relationship model example* for a full example.

`OneToOneField` fields also accept one specific, optional `parent_link` argument described in the *model field reference*.

`OneToOneField` classes used to automatically become the primary key on a model. This is no longer true (although you can manually pass in the `primary_key` argument if you like). Thus, it's now possible to have multiple fields of type `OneToOneField` on a single model.

Models across files

It's perfectly OK to relate a model to one from another app. To do this, import the related model at the top of the file where your model is defined. Then, just refer to the other model class wherever needed. For example:

```
from geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(ZipCode)
```

Field name restrictions

Django places only two restrictions on model field names:

1. A field name cannot be a Python reserved word, because that would result in a Python syntax error. For example:

   ```
   class Example(models.Model):
       pass = models.IntegerField() # 'pass' is a reserved word!
   ```

2. A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works. For example:

   ```
   class Example(models.Model):
       foo__bar = models.IntegerField() # 'foo__bar' has two underscores!
   ```

These limitations can be worked around, though, because your field name doesn't necessarily have to match your database column name. See the `db_column` option.

SQL reserved words, such as `join`, `where` or `select`, *are* allowed as model field names, because Django escapes all database table names and column names in every underlying SQL query. It uses the quoting syntax of your particular database engine.

Custom field types

If one of the existing model fields cannot be used to fit your purposes, or if you wish to take advantage of some less common database column types, you can create your own field class. Full coverage of creating your own fields is provided in *Writing custom model fields*.

Meta options

Give your model metadata by using an inner `class Meta`, like so:

```
class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

Model metadata is "anything that's not a field", such as ordering options (`ordering`), database table name (`db_table`), or human-readable singular and plural names (`verbose_name` and `verbose_name_plural`). None are required, and adding `class Meta` to a model is completely optional.

A complete list of all possible `Meta` options can be found in the *model option reference*.

Model methods

Define custom methods on a model to add custom "row-level" functionality to your objects. Whereas `Manager` methods are intended to do "table-wide" things, model methods should act on a particular model instance.

This is a valuable technique for keeping business logic in one place – the model.

For example, this model has a few custom methods:

```
from django.contrib.localflavor.us.models import USStateField

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()
    address = models.CharField(max_length=100)
    city = models.CharField(max_length=50)
    state = USStateField() # Yes, this is America-centric...

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if datetime.date(1945, 8, 1) <= self.birth_date <= datetime.date(1964, 12, 31):
            return "Baby boomer"
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        return "Post-boomer"

    def is_midwestern(self):
        "Returns True if this person is from the Midwest."
        return self.state in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')

    def _get_full_name(self):
        "Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
    full_name = property(_get_full_name)
```

The last method in this example is a *property*.

The *model instance reference* has a complete list of *methods automatically given to each model*. You can override most of these – see overriding predefined model methods, below – but there are a couple that you'll almost always want to define:

`__unicode__()` A Python "magic method" that returns a unicode "representation" of any object. This is what Python and Django will use whenever a model instance needs to be coerced and displayed as a plain string. Most notably, this happens when you display an object in an interactive console or in the admin.

You'll always want to define this method; the default isn't very helpful at all.

`get_absolute_url()` This tells Django how to calculate the URL for an object. Django uses this in its admin interface, and any time it needs to figure out a URL for an object.

Any object that has a URL that uniquely identifies it should define this method.

Overriding predefined model methods

There's another set of *model methods* that encapsulate a bunch of database behavior that you'll want to customize. In particular you'll often want to change the way `save()` and `delete()` work.

You're free to override these methods (and any other model method) to alter behavior.

A classic use-case for overriding the built-in methods is if you want something to happen whenever you save an object. For example (see `save()` for documentation of the parameters it accepts):

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
        do_something_else()
```

You can also prevent saving:

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super(Blog, self).save(*args, **kwargs) # Call the "real" save() method.
```

It's important to remember to call the superclass method – that's that `super(Blog, self).save(*args, **kwargs)` business – to ensure that the object still gets saved into the database. If you forget to call the superclass method, the default behavior won't happen and the database won't get touched.

It's also important that you pass through the arguments that can be passed to the model method – that's what the `*args, **kwargs` bit does. Django will, from time to time, extend the capabilities of built-in model methods, adding new arguments. If you use `*args, **kwargs` in your method definitions, you are guaranteed that your code will automatically support those arguments when they are added.

Overriding Delete

Note that the `delete()` method for an object is not necessarily called when *deleting objects in bulk using a QuerySet*. To ensure customized delete logic gets executed, you can use `pre_delete` and/or `post_delete` signals.

Executing custom SQL

Another common pattern is writing custom SQL statements in model methods and module-level methods. For more details on using raw SQL, see the documentation on *using raw SQL*.

Model inheritance

Model inheritance in Django works almost identically to the way normal class inheritance works in Python. The only decision you have to make is whether you want the parent models to be models in their own right (with their own

database tables), or if the parents are just holders of common information that will only be visible through the child models.

There are three styles of inheritance that are possible in Django.

1. Often, you will just want to use the parent class to hold information that you don't want to have to type out for each child model. This class isn't going to ever be used in isolation, so *Abstract base classes* are what you're after.
2. If you're subclassing an existing model (perhaps something from another application entirely) and want each model to have its own database table, *Multi-table inheritance* is the way to go.
3. Finally, if you only want to modify the Python-level behavior of a model, without changing the models fields in any way, you can use *Proxy models*.

Abstract base classes

Abstract base classes are useful when you want to put some common information into a number of other models. You write your base class and put `abstract=True` in the *Meta* class. This model will then not be used to create any database table. Instead, when it is used as a base class for other models, its fields will be added to those of the child class. It is an error to have fields in the abstract base class with the same name as those in the child (and Django will raise an exception).

An example:

```
class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

The `Student` model will have three fields: `name`, `age` and `home_group`. The `CommonInfo` model cannot be used as a normal Django model, since it is an abstract base class. It does not generate a database table or have a manager, and cannot be instantiated or saved directly.

For many uses, this type of model inheritance will be exactly what you want. It provides a way to factor out common information at the Python level, whilst still only creating one database table per child model at the database level.

`Meta` inheritance When an abstract base class is created, Django makes any *Meta* inner class you declared in the base class available as an attribute. If a child class does not declare its own *Meta* class, it will inherit the parent's *Meta*. If the child wants to extend the parent's *Meta* class, it can subclass it. For example:

```
class CommonInfo(models.Model):
    ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

Django does make one adjustment to the *Meta* class of an abstract base class: before installing the *Meta* attribute, it sets `abstract=False`. This means that children of abstract base classes don't automatically become abstract classes themselves. Of course, you can make an abstract base class that inherits from another abstract base class. You just need to remember to explicitly set `abstract=True` each time.

Some attributes won't make sense to include in the *Meta* class of an abstract base class. For example, including `db_table` would mean that all the child classes (the ones that don't specify their own *Meta*) would use the same database table, which is almost certainly not what you want.

Be careful with `related_name` If you are using the `related_name` attribute on a `ForeignKey` or `ManyToManyField`, you must always specify a *unique* reverse name for the field. This would normally cause a problem in abstract base classes, since the fields on this class are included into each of the child classes, with exactly the same values for the attributes (including `related_name`) each time.

To work around this problem, when you are using `related_name` in an abstract base class (only), part of the name should contain `'%(app_label)s'` and `'%(class)s'`.

- `'%(class)s'` is replaced by the lower-cased name of the child class that the field is used in.
- `'%(app_label)s'` is replaced by the lower-cased name of the app the child class is contained within. Each installed application name must be unique and the model class names within each app must also be unique, therefore the resulting name will end up being different.

For example, given an app `common/models.py`:

```
class Base(models.Model):
    m2m = models.ManyToManyField(OtherModel, related_name="%(app_label)s_%(class)s_related")

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

Along with another app `rare/models.py`:

```
from common.models import Base

class ChildB(Base):
    pass
```

The reverse name of the `common.ChildA.m2m` field will be `common_childa_related`, whilst the reverse name of the `common.ChildB.m2m` field will be `common_childb_related`, and finally the reverse name of the `rare.ChildB.m2m` field will be `rare_childb_related`. It is up to you how you use the `'%(class)s'` and `'%(app_label)s` portion to construct your related name, but if you forget to use it, Django will raise errors when you validate your models (or run `syncdb`).

If you don't specify a `related_name` attribute for a field in an abstract base class, the default reverse name will be the name of the child class followed by `'_set'`, just as it normally would be if you'd declared the field directly on the child class. For example, in the above code, if the `related_name` attribute was omitted, the reverse name for the `m2m` field would be `childa_set` in the `ChildA` case and `childb_set` for the `ChildB` field.

Multi-table inheritance

The second type of model inheritance supported by Django is when each model in the hierarchy is a model all by itself. Each model corresponds to its own database table and can be queried and created individually. The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created `OneToOneField`). For example:

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField()
    serves_pizza = models.BooleanField()
```

All of the fields of `Place` will also be available in `Restaurant`, although the data will reside in a different database table. So these are both possible:

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

If you have a `Place` that is also a `Restaurant`, you can get from the `Place` object to the `Restaurant` object by using the lower-case version of the model name:

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

However, if `p` in the above example was *not* a `Restaurant` (it had been created directly as a `Place` object or was the parent of some other class), referring to `p.restaurant` would raise a Restaurant.DoesNotExist exception.

`Meta` and multi-table inheritance In the multi-table inheritance situation, it doesn't make sense for a child class to inherit from its parent's *Meta* class. All the *Meta* options have already been applied to the parent class and applying them again would normally only lead to contradictory behavior (this is in contrast with the abstract base class case, where the base class doesn't exist in its own right).

So a child model does not have access to its parent's *Meta* class. However, there are a few limited cases where the child inherits behavior from the parent: if the child does not specify an `ordering` attribute or a `get_latest_by` attribute, it will inherit these from its parent.

If the parent has an ordering and you don't want the child to have any natural ordering, you can explicitly disable it:

```
class ChildModel(ParentModel):
    ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

Inheritance and reverse relations Because multi-table inheritance uses an implicit `OneToOneField` to link the child and the parent, it's possible to move from the parent down to the child, as in the above example. However, this uses up the name that is the default `related_name` value for `ForeignKey` and `ManyToManyField` relations. If you are putting those types of relations on a subclass of another model, you **must** specify the `related_name` attribute on each such field. If you forget, Django will raise an error when you run `validate` or `syncdb`.

For example, using the above `Place` class again, let's create another subclass with a `ManyToManyField`:

```
class Supplier(Place):
    # Must specify related_name on all relations.
    customers = models.ManyToManyField(Restaurant, related_name='provider')
```

Specifying the parent link field As mentioned, Django will automatically create a `OneToOneField` linking your child class back any non-abstract parent models. If you want to control the name of the attribute linking back to the parent, you can create your own `OneToOneField` and set `parent_link=True` to indicate that your field is the link back to the parent class.

Proxy models

When using *multi-table inheritance*, a new database table is created for each subclass of a model. This is usually the desired behavior, since the subclass needs a place to store any additional data fields that are not present on the base class. Sometimes, however, you only want to change the Python behavior of a model – perhaps to change the default manager, or add a new method.

This is what proxy model inheritance is for: creating a *proxy* for the original model. You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model. The difference is that you can change things like the default model ordering or the default manager in the proxy, without having to alter the original.

Proxy models are declared like normal models. You tell Django that it's a proxy model by setting the `proxy` attribute of the `Meta` class to `True`.

For example, suppose you want to add a method to the standard `User` model that will be used in your templates. You can do it like this:

```
from django.contrib.auth.models import User

class MyUser(User):
    class Meta:
        proxy = True

    def do_something(self):
        ...
```

The `MyUser` class operates on the same database table as its parent `User` class. In particular, any new instances of `User` will also be accessible through `MyUser`, and vice-versa:

```
>>> u = User.objects.create(username="foobar")
>>> MyUser.objects.get(username="foobar")
<MyUser: foobar>
```

You could also use a proxy model to define a different default ordering on a model. The standard `User` model has no ordering defined on it (intentionally; sorting is expensive and we don't want to do it all the time when we fetch users). You might want to regularly order by the `username` attribute when you use the proxy. This is easy:

```
class OrderedUser(User):
    class Meta:
        ordering = ["username"]
        proxy = True
```

Now normal `User` queries will be unordered and `OrderedUser` queries will be ordered by `username`.

QuerySets still return the model that was requested There is no way to have Django return, say, a `MyUser` object whenever you query for `User` objects. A queryset for `User` objects will return those types of objects. The

whole point of proxy objects is that code relying on the original `User` will use those and your own code can use the extensions you included (that no other code is relying on anyway). It is not a way to replace the `User` (or any other) model everywhere with something of your own creation.

Base class restrictions A proxy model must inherit from exactly one non-abstract model class. You can't inherit from multiple non-abstract models as the proxy model doesn't provide any connection between the rows in the different database tables. A proxy model can inherit from any number of abstract model classes, providing they do *not* define any model fields.

Proxy models inherit any `Meta` options that they don't define from their non-abstract model parent (the model they are proxying for).

Proxy model managers If you don't specify any model managers on a proxy model, it inherits the managers from its model parents. If you define a manager on the proxy model, it will become the default, although any managers defined on the parent classes will still be available.

Continuing our example from above, you could change the default manager used when you query the `User` model like this:

```
class NewManager(models.Manager):
    ...

class MyUser(User):
    objects = NewManager()

    class Meta:
        proxy = True
```

If you wanted to add a new manager to the Proxy, without replacing the existing default, you can use the techniques described in the *custom manager* documentation: create a base class containing the new managers and inherit that after the primary base class:

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True

class MyUser(User, ExtraManagers):
    class Meta:
        proxy = True
```

You probably won't need to do this very often, but, when you do, it's possible.

Differences between proxy inheritance and unmanaged models Proxy model inheritance might look fairly similar to creating an unmanaged model, using the `managed` attribute on a model's `Meta` class. The two alternatives are not quite the same and it's worth considering which one you should use.

One difference is that you can (and, in fact, must unless you want an empty model) specify model fields on models with `Meta.managed=False`. You could, with careful setting of `Meta.db_table` create an unmanaged model that shadowed an existing model and add Python methods to it. However, that would be very repetitive and fragile as you need to keep both copies synchronized if you make any changes.

The other difference that is more important for proxy models, is how model managers are handled. Proxy models are intended to behave exactly like the model they are proxying for. So they inherit the parent model's managers, including the default manager. In the normal multi-table model inheritance case, children do not inherit managers from their

parents as the custom managers aren't always appropriate when extra fields are involved. The *manager documentation* has more details about this latter case.

When these two features were implemented, attempts were made to squash them into a single option. It turned out that interactions with inheritance, in general, and managers, in particular, made the API very complicated and potentially difficult to understand and use. It turned out that two options were needed in any case, so the current separation arose.

So, the general rules are:

1. If you are mirroring an existing model or database table and don't want all the original database table columns, use `Meta.managed=False`. That option is normally useful for modeling database views and tables not under the control of Django.
2. If you are wanting to change the Python-only behavior of a model, but keep all the same fields as in the original, use `Meta.proxy=True`. This sets things up so that the proxy model is an exact copy of the storage structure of the original model when data is saved.

Multiple inheritance

Just as with Python's subclassing, it's possible for a Django model to inherit from multiple parent models. Keep in mind that normal Python name resolution rules apply. The first base class that a particular name (e.g. *Meta*) appears in will be the one that is used; for example, this means that if multiple parents contain a *Meta* class, only the first one is going to be used, and all others will be ignored.

Generally, you won't need to inherit from multiple parents. The main use-case where this is useful is for "mix-in" classes: adding a particular extra field or method to every class that inherits the mix-in. Try to keep your inheritance hierarchies as simple and straightforward as possible so that you won't have to struggle to work out where a particular piece of information is coming from.

Field name "hiding" is not permitted

In normal Python class inheritance, it is permissible for a child class to override any attribute from the parent class. In Django, this is not permitted for attributes that are `Field` instances (at least, not at the moment). If a base class has a field called `author`, you cannot create another model field called `author` in any class that inherits from that base class.

Overriding fields in a parent model leads to difficulties in areas such as initializing new instances (specifying which field is being initialized in `Model.__init__`) and serialization. These are features which normal Python class inheritance doesn't have to deal with in quite the same way, so the difference between Django model inheritance and Python class inheritance isn't arbitrary.

This restriction only applies to attributes which are `Field` instances. Normal Python attributes can be overridden if you wish. It also only applies to the name of the attribute as Python sees it: if you are manually specifying the database column name, you can have the same column name appearing in both a child and an ancestor model for multi-table inheritance (they are columns in two different database tables).

Django will raise a `FieldError` if you override any model field in any ancestor model.

3.2.2 Making queries

Once you've created your *data models*, Django automatically gives you a database-abstraction API that lets you create, retrieve, update and delete objects. This document explains how to use this API. Refer to the *data model reference* for full details of all the various model lookup options.

Throughout this guide (and in the reference), we'll refer to the following models, which comprise a Weblog application:

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __unicode__(self):
        return self.name

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

    def __unicode__(self):
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateTimeField()
    mod_date = models.DateTimeField()
    authors = models.ManyToManyField(Author)
    n_comments = models.IntegerField()
    n_pingbacks = models.IntegerField()
    rating = models.IntegerField()

    def __unicode__(self):
        return self.headline
```

Creating objects

To represent database-table data in Python objects, Django uses an intuitive system: A model class represents a database table, and an instance of that class represents a particular record in the database table.

To create an object, instantiate it using keyword arguments to the model class, then call `save()` to save it to the database.

You import the model class from wherever it lives on the Python path, as you may expect. (We point this out here because previous Django versions required funky model importing.)

Assuming models live in a file `mysite/blog/models.py`, here's an example:

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

This performs an `INSERT` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call `save()`.

The `save()` method has no return value.

See Also:

`save()` takes a number of advanced options not described here. See the documentation for `save()` for complete details.

To create and save an object in a single step, use the `create()` method.

Saving changes to objects

To save changes to an object that's already in the database, use `save()`.

Given a `Blog` instance `b5` that has already been saved to the database, this example changes its name and updates its record in the database:

```
>> b5.name = 'New name'
>> b5.save()
```

This performs an `UPDATE` SQL statement behind the scenes. Django doesn't hit the database until you explicitly call `save()`.

Saving `ForeignKey` and `ManyToManyField` fields

Updating a `ForeignKey` field works exactly the same way as saving a normal field – simply assign an object of the right type to the field in question. This example updates the `blog` attribute of an `Entry` instance `entry`:

```
>>> from blog.models import Entry
>>> entry = Entry.objects.get(pk=1)
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

Updating a `ManyToManyField` works a little differently – use the `add()` method on the field to add a record to the relation. This example adds the `Author` instance `joe` to the `entry` object:

```
>>> from blog.models import Author
>>> joe = Author.objects.create(name="Joe")
>>> entry.authors.add(joe)
```

To add multiple records to a `ManyToManyField` in one go, include multiple arguments in the call to `add()`, like this:

```
>>> john = Author.objects.create(name="John")
>>> paul = Author.objects.create(name="Paul")
>>> george = Author.objects.create(name="George")
>>> ringo = Author.objects.create(name="Ringo")
>>> entry.authors.add(john, paul, george, ringo)
```

Django will complain if you try to assign or add an object of the wrong type.

Retrieving objects

To retrieve objects from your database, construct a `QuerySet` via a `Manager` on your model class.

A `QuerySet` represents a collection of objects from your database. It can have zero, one or many *filters* – criteria that narrow down the collection based on given parameters. In SQL terms, a `QuerySet` equates to a `SELECT` statement, and a filter is a limiting clause such as `WHERE` or `LIMIT`.

You get a `QuerySet` by using your model's `Manager`. Each model has at least one `Manager`, and it's called `objects` by default. Access it directly via the model class, like so:

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
```

```
    ...
AttributeError: "Manager isn't accessible via Blog instances."
```

Note: `Managers` are accessible only via model classes, rather than from model instances, to enforce a separation between "table-level" operations and "record-level" operations.

The `Manager` is the main source of `QuerySets` for a model. It acts as a "root" `QuerySet` that describes all objects in the model's database table. For example, `Blog.objects` is the initial `QuerySet` that contains all `Blog` objects in the database.

Retrieving all objects

The simplest way to retrieve objects from a table is to get all of them. To do this, use the `all()` method on a `Manager`:

```
>>> all_entries = Entry.objects.all()
```

The `all()` method returns a `QuerySet` of all the objects in the database.

(If `Entry.objects` is a `QuerySet`, why can't we just do `Entry.objects`? That's because `Entry.objects`, the root `QuerySet`, is a special case that cannot be evaluated. The `all()` method returns a `QuerySet` that *can* be evaluated.)

Retrieving specific objects with filters

The root `QuerySet` provided by the `Manager` describes all objects in the database table. Usually, though, you'll need to select only a subset of the complete set of objects.

To create such a subset, you refine the initial `QuerySet`, adding filter conditions. The two most common ways to refine a `QuerySet` are:

`filter(**kwargs)` Returns a new `QuerySet` containing objects that match the given lookup parameters.

`exclude(**kwargs)` Returns a new `QuerySet` containing objects that do *not* match the given lookup parameters.

The lookup parameters (`**kwargs` in the above function definitions) should be in the format described in Field lookups below.

For example, to get a `QuerySet` of blog entries from the year 2006, use `filter()` like so:

```
Entry.objects.filter(pub_date__year=2006)
```

We don't have to add an `all()` – `Entry.objects.all().filter(...)`. That would still work, but you only need `all()` when you want all objects from the root `QuerySet`.

Chaining filters The result of refining a `QuerySet` is itself a `QuerySet`, so it's possible to chain refinements together. For example:

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.now()
... ).filter(
...     pub_date__gte=datetime(2005, 1, 1)
... )
```

This takes the initial `QuerySet` of all entries in the database, adds a filter, then an exclusion, then another filter. The final result is a `QuerySet` containing all entries with a headline that starts with "What", that were published between January 1, 2005, and the current day.

Filtered QuerySets are unique Each time you refine a `QuerySet`, you get a brand-new `QuerySet` that is in no way bound to the previous `QuerySet`. Each refinement creates a separate and distinct `QuerySet` that can be stored, used and reused.

Example:

```
>> q1 = Entry.objects.filter(headline__startswith="What")
>> q2 = q1.exclude(pub_date__gte=datetime.now())
>> q3 = q1.filter(pub_date__gte=datetime.now())
```

These three `QuerySets` are separate. The first is a base `QuerySet` containing all entries that contain a headline starting with "What". The second is a subset of the first, with an additional criteria that excludes records whose `pub_date` is greater than now. The third is a subset of the first, with an additional criteria that selects only the records whose `pub_date` is greater than now. The initial `QuerySet` (`q1`) is unaffected by the refinement process.

QuerySets are lazy `QuerySets` are lazy – the act of creating a `QuerySet` doesn't involve any database activity. You can stack filters together all day long, and Django won't actually run the query until the `QuerySet` is *evaluated*. Take a look at this example:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.now())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

Though this looks like three database hits, in fact it hits the database only once, at the last line (`print(q)`). In general, the results of a `QuerySet` aren't fetched from the database until you "ask" for them. When you do, the `QuerySet` is *evaluated* by accessing the database. For more details on exactly when evaluation takes place, see *When QuerySets are evaluated*.

Retrieving a single object with get

`filter()` will always give you a `QuerySet`, even if only a single object matches the query - in this case, it will be a `QuerySet` containing a single element.

If you know there is only one object that matches your query, you can use the `get()` method on a *Manager* which returns the object directly:

```
>>> one_entry = Entry.objects.get(pk=1)
```

You can use any query expression with `get()`, just like with `filter()` - again, see Field lookups below.

Note that there is a difference between using `get()`, and using `filter()` with a slice of `[0]`. If there are no results that match the query, `get()` will raise a `DoesNotExist` exception. This exception is an attribute of the model class that the query is being performed on - so in the code above, if there is no `Entry` object with a primary key of 1, Django will raise `Entry.DoesNotExist`.

Similarly, Django will complain if more than one item matches the `get()` query. In this case, it will raise `MultipleObjectsReturned`, which again is an attribute of the model class itself.

Other QuerySet methods

Most of the time you'll use `all()`, `get()`, `filter()` and `exclude()` when you need to look up objects from the database. However, that's far from all there is; see the *QuerySet API Reference* for a complete list of all the various `QuerySet` methods.

Limiting QuerySets

Use a subset of Python's array-slicing syntax to limit your `QuerySet` to a certain number of results. This is the equivalent of SQL's `LIMIT` and `OFFSET` clauses.

For example, this returns the first 5 objects (`LIMIT 5`):

```
>>> Entry.objects.all()[:5]
```

This returns the sixth through tenth objects (`OFFSET 5 LIMIT 5`):

```
>>> Entry.objects.all()[5:10]
```

Negative indexing (i.e. `Entry.objects.all()[-1]`) is not supported.

Generally, slicing a `QuerySet` returns a new `QuerySet` – it doesn't evaluate the query. An exception is if you use the "step" parameter of Python slice syntax. For example, this would actually execute the query in order to return a list of every *second* object of the first 10:

```
>>> Entry.objects.all()[:10:2]
```

To retrieve a *single* object rather than a list (e.g. `SELECT foo FROM bar LIMIT 1`), use a simple index instead of a slice. For example, this returns the first `Entry` in the database, after ordering entries alphabetically by headline:

```
>>> Entry.objects.order_by('headline')[0]
```

This is roughly equivalent to:

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

Note, however, that the first of these will raise `IndexError` while the second will raise `DoesNotExist` if no objects match the given criteria. See `get()` for more details.

Field lookups

Field lookups are how you specify the meat of an SQL `WHERE` clause. They're specified as keyword arguments to the `QuerySet` methods `filter()`, `exclude()` and `get()`.

Basic lookups keyword arguments take the form `field__lookuptype=value`. (That's a double-underscore). For example:

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

How this is possible

Python has the ability to define functions that accept arbitrary name-value arguments whose names and values are evaluated at runtime. For more information, see Keyword Arguments in the official Python tutorial.

Changed in version 1.4: The field specified in a lookup has to be the name of a model field. There's one exception though, in case of a `ForeignKey` you can specify the field name suffixed with `_id`. In this case, the value parameter is expected to contain the raw value of the foreign model's primary key. For example::

```
>>> Entry.objects.filter(blog_id__exact=4)
```

If you pass an invalid keyword argument, a lookup function will raise `TypeError`.

The database API supports about two dozen lookup types; a complete reference can be found in the *field lookup reference*. To give you a taste of what's available, here's some of the more common lookups you'll probably use:

`exact` An "exact" match. For example:

```
>>> Entry.objects.get(headline__exact="Man bites dog")
```

Would generate SQL along these lines:

```
SELECT ... WHERE headline = 'Man bites dog';
```

If you don't provide a lookup type – that is, if your keyword argument doesn't contain a double underscore – the lookup type is assumed to be `exact`.

For example, the following two statements are equivalent:

```
>>> Blog.objects.get(id__exact=14)  # Explicit form
>>> Blog.objects.get(id=14)         # __exact is implied
```

This is for convenience, because `exact` lookups are the common case.

`iexact` A case-insensitive match. So, the query:

```
>>> Blog.objects.get(name__iexact="beatles blog")
```

Would match a `Blog` titled "Beatles Blog", "beatles blog", or even "BeAtlES blOG".

`contains` Case-sensitive containment test. For example:

```
Entry.objects.get(headline__contains='Lennon')
```

Roughly translates to this SQL:

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

Note this will match the headline `'Today Lennon honored'` but not `'today lennon honored'`.

There's also a case-insensitive version, `icontains`.

`startswith`, `endswith` Starts-with and ends-with search, respectively. There are also case-insensitive versions called `istartswith` and `iendswith`.

Again, this only scratches the surface. A complete reference can be found in the *field lookup reference*.

Lookups that span relationships

Django offers a powerful and intuitive way to "follow" relationships in lookups, taking care of the SQL `JOIN`s for you automatically, behind the scenes. To span a relationship, just use the field name of related fields across models, separated by double underscores, until you get to the field you want.

This example retrieves all `Entry` objects with a `Blog` whose `name` is `'Beatles Blog'`:

```
>>> Entry.objects.filter(blog__name__exact='Beatles Blog')
```

This spanning can be as deep as you'd like.

It works backwards, too. To refer to a "reverse" relationship, just use the lowercase name of the model.

This example retrieves all `Blog` objects which have at least one `Entry` whose `headline` contains `'Lennon'`:

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

If you are filtering across multiple relationships and one of the intermediate models doesn't have a value that meets the filter condition, Django will treat it as if there is an empty (all values are `NULL`), but valid, object there. All this means is that no error will be raised. For example, in this filter:

```
Blog.objects.filter(entry__authors__name='Lennon')
```

(if there was a related `Author` model), if there was no `author` associated with an entry, it would be treated as if there was also no `name` attached, rather than raising an error because of the missing `author`. Usually this is exactly what you want to have happen. The only case where it might be confusing is if you are using `isnull`. Thus:

```
Blog.objects.filter(entry__authors__name__isnull=True)
```

will return `Blog` objects that have an empty `name` on the `author` and also those which have an empty `author` on the `entry`. If you don't want those latter objects, you could write:

```
Blog.objects.filter(entry__authors__isnull=False,
        entry__authors__name__isnull=True)
```

Spanning multi-valued relationships When you are filtering an object based on a `ManyToManyField` or a reverse `ForeignKey`, there are two different sorts of filter you may be interested in. Consider the `Blog`/`Entry` relationship (`Blog` to `Entry` is a one-to-many relation). We might be interested in finding blogs that have an entry which has both *"Lennon"* in the headline and was published in 2008. Or we might want to find blogs that have an entry with *"Lennon"* in the headline as well as an entry that was published in 2008. Since there are multiple entries associated with a single `Blog`, both of these queries are possible and make sense in some situations.

The same type of situation arises with a `ManyToManyField`. For example, if an `Entry` has a `ManyToManyField` called `tags`, we might want to find entries linked to tags called *"music"* and *"bands"* or we might want an entry that contains a tag with a name of *"music"* and a status of *"public"*.

To handle both of these situations, Django has a consistent way of processing `filter()` and `exclude()` calls. Everything inside a single `filter()` call is applied simultaneously to filter out items matching all those requirements. Successive `filter()` calls further restrict the set of objects, but for multi-valued relations, they apply to any object linked to the primary model, not necessarily those objects that were selected by an earlier `filter()` call.

That may sound a bit confusing, so hopefully an example will clarify. To select all blogs that contain entries with both *"Lennon"* in the headline and that were published in 2008 (the same entry satisfying both conditions), we would write:

```
Blog.objects.filter(entry__headline__contains='Lennon',
        entry__pub_date__year=2008)
```

To select all blogs that contain an entry with *"Lennon"* in the headline **as well as** an entry that was published in 2008, we would write:

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(
        entry__pub_date__year=2008)
```

In this second example, the first filter restricted the queryset to all those blogs linked to that particular type of entry. The second filter restricted the set of blogs *further* to those that are also linked to the second type of entry. The entries

select by the second filter may or may not be the same as the entries in the first filter. We are filtering the `Blog` items with each filter statement, not the `Entry` items.

All of this behavior also applies to `exclude()`: all the conditions in a single `exclude()` statement apply to a single instance (if those conditions are talking about the same multi-valued relation). Conditions in subsequent `filter()` or `exclude()` calls that refer to the same relation may end up filtering on different linked objects.

Filters can reference fields on the model

In the examples given so far, we have constructed filters that compare the value of a model field with a constant. But what if you want to compare the value of a model field with another field on the same model?

Django provides the *F() expressions* to allow such comparisons. Instances of `F()` act as a reference to a model field within a query. These references can then be used in query filters to compare the values of two different fields on the same model instance.

For example, to find a list of all blog entries that have had more comments than pingbacks, we construct an `F()` object to reference the pingback count, and use that `F()` object in the query:

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks'))
```

Django supports the use of addition, subtraction, multiplication, division and modulo arithmetic with `F()` objects, both with constants and with other `F()` objects. To find all the blog entries with more than *twice* as many comments as pingbacks, we modify the query:

```
>>> Entry.objects.filter(n_comments__gt=F('n_pingbacks') * 2)
```

To find all the entries where the rating of the entry is less than the sum of the pingback count and comment count, we would issue the query:

```
>>> Entry.objects.filter(rating__lt=F('n_comments') + F('n_pingbacks'))
```

You can also use the double underscore notation to span relationships in an `F()` object. An `F()` object with a double underscore will introduce any joins needed to access the related object. For example, to retrieve all the entries where the author's name is the same as the blog name, we could issue the query:

```
>>> Entry.objects.filter(authors__name=F('blog__name'))
```

New in version 1.3: *Please see the release notes* For date and date/time fields, you can add or subtract a `timedelta` object. The following would return all entries that were modified more than 3 days after they were published:

```
>>> from datetime import timedelta
>>> Entry.objects.filter(mod_date__gt=F('pub_date') + timedelta(days=3))
```

The pk lookup shortcut

For convenience, Django provides a `pk` lookup shortcut, which stands for "primary key".

In the example `Blog` model, the primary key is the `id` field, so these three statements are equivalent:

```
>>> Blog.objects.get(id__exact=14) # Explicit form
>>> Blog.objects.get(id=14) # __exact is implied
>>> Blog.objects.get(pk=14) # pk implies id__exact
```

The use of `pk` isn't limited to `__exact` queries – any query term can be combined with `pk` to perform a query on the primary key of a model:

```
# Get blogs entries with id 1, 4 and 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Get all blog entries with id > 14
>>> Blog.objects.filter(pk__gt=14)
```

`pk` lookups also work across joins. For example, these three statements are equivalent:

```
>>> Entry.objects.filter(blog__id__exact=3) # Explicit form
>>> Entry.objects.filter(blog__id=3)        # __exact is implied
>>> Entry.objects.filter(blog__pk=3)        # __pk implies __id__exact
```

Escaping percent signs and underscores in LIKE statements

The field lookups that equate to `LIKE` SQL statements (`iexact`, `contains`, `icontains`, `startswith`, `istartswith`, `endswith` and `iendswith`) will automatically escape the two special characters used in `LIKE` statements – the percent sign and the underscore. (In a `LIKE` statement, the percent sign signifies a multiple-character wildcard and the underscore signifies a single-character wildcard.)

This means things should work intuitively, so the abstraction doesn't leak. For example, to retrieve all the entries that contain a percent sign, just use the percent sign as any other character:

```
>>> Entry.objects.filter(headline__contains='%')
```

Django takes care of the quoting for you; the resulting SQL will look something like this:

```
SELECT ... WHERE headline LIKE '%\%%';
```

Same goes for underscores. Both percentage signs and underscores are handled for you transparently.

Caching and QuerySets

Each `QuerySet` contains a cache, to minimize database access. It's important to understand how it works, in order to write the most efficient code.

In a newly created `QuerySet`, the cache is empty. The first time a `QuerySet` is evaluated – and, hence, a database query happens – Django saves the query results in the `QuerySet`'s cache and returns the results that have been explicitly requested (e.g., the next element, if the `QuerySet` is being iterated over). Subsequent evaluations of the `QuerySet` reuse the cached results.

Keep this caching behavior in mind, because it may bite you if you don't use your `QuerySet`s correctly. For example, the following will create two `QuerySet`s, evaluate them, and throw them away:

```
>>> print([e.headline for e in Entry.objects.all()])
>>> print([e.pub_date for e in Entry.objects.all()])
```

That means the same database query will be executed twice, effectively doubling your database load. Also, there's a possibility the two lists may not include the same database records, because an `Entry` may have been added or deleted in the split second between the two requests.

To avoid this problem, simply save the `QuerySet` and reuse it:

```
>>> queryset = Entry.objects.all()
>>> print([p.headline for p in queryset]) # Evaluate the query set.
>>> print([p.pub_date for p in queryset]) # Re-use the cache from the evaluation.
```

Complex lookups with Q objects

Keyword argument queries – in `filter()`, etc. – are "AND"ed together. If you need to execute more complex queries (for example, queries with `OR` statements), you can use `Q` objects.

A `Q` object (`django.db.models.Q`) is an object used to encapsulate a collection of keyword arguments. These keyword arguments are specified as in "Field lookups" above.

For example, this `Q` object encapsulates a single `LIKE` query:

```
from django.db.models import Q
Q(question__startswith='What')
```

`Q` objects can be combined using the `&` and `|` operators. When an operator is used on two `Q` objects, it yields a new `Q` object.

For example, this statement yields a single `Q` object that represents the "OR" of two `"question__startswith"` queries:

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

This is equivalent to the following SQL `WHERE` clause:

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

You can compose statements of arbitrary complexity by combining `Q` objects with the `&` and `|` operators and use parenthetical grouping. Also, `Q` objects can be negated using the `~` operator, allowing for combined lookups that combine both a normal query and a negated (`NOT`) query:

```
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

Each lookup function that takes keyword-arguments (e.g. `filter()`, `exclude()`, `get()`) can also be passed one or more `Q` objects as positional (not-named) arguments. If you provide multiple `Q` object arguments to a lookup function, the arguments will be "AND"ed together. For example:

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... roughly translates into the SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
    AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

Lookup functions can mix the use of `Q` objects and keyword arguments. All arguments provided to a lookup function (be they keyword arguments or `Q` objects) are "AND"ed together. However, if a `Q` object is provided, it must precede the definition of any keyword arguments. For example:

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who')
```

... would be a valid query, equivalent to the previous example; but:

```
# INVALID QUERY
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)))
```

... would not be valid.

See Also:

The OR lookups examples in the Django unit tests show some possible uses of `Q`.

Comparing objects

To compare two model instances, just use the standard Python comparison operator, the double equals sign: `==`. Behind the scenes, that compares the primary key values of two models.

Using the `Entry` example above, the following two statements are equivalent:

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

If a model's primary key isn't called `id`, no problem. Comparisons will always use the primary key, whatever it's called. For example, if a model's primary key field is called `name`, these two statements are equivalent:

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

Deleting objects

The delete method, conveniently, is named `delete()`. This method immediately deletes the object and has no return value. Example:

```
e.delete()
```

You can also delete objects in bulk. Every `QuerySet` has a `delete()` method, which deletes all members of that `QuerySet`.

For example, this deletes all `Entry` objects with a `pub_date` year of 2005:

```
Entry.objects.filter(pub_date__year=2005).delete()
```

Keep in mind that this will, whenever possible, be executed purely in SQL, and so the `delete()` methods of individual object instances will not necessarily be called during the process. If you've provided a custom `delete()` method on a model class and want to ensure that it is called, you will need to "manually" delete instances of that model (e.g., by iterating over a `QuerySet` and calling `delete()` on each object individually) rather than using the bulk `delete()` method of a `QuerySet`.

When Django deletes an object, by default it emulates the behavior of the SQL constraint `ON DELETE CASCADE` – in other words, any objects which had foreign keys pointing at the object to be deleted will be deleted along with it. For example:

```
b = Blog.objects.get(pk=1)
# This will delete the Blog and all of its Entry objects.
b.delete()
```

New in version 1.3: This cascade behavior is customizable via the `on_delete` argument to the `ForeignKey`. Note that `delete()` is the only `QuerySet` method that is not exposed on a `Manager` itself. This is a safety mechanism to prevent you from accidentally requesting `Entry.objects.delete()`, and deleting *all* the entries. If you *do* want to delete all the objects, then you have to explicitly request a complete query set:

```
Entry.objects.all().delete()
```

Copying model instances

Although there is no built-in method for copying model instances, it is possible to easily create new instance with all fields' values copied. In the simplest case, you can just set `pk` to `None`. Using our blog example:

```
blog = Blog(name='My blog', tagline='Blogging is easy')
blog.save() # post.pk == 1

blog.pk = None
blog.save() # post.pk == 2
```

Things get more complicated if you use inheritance. Consider a subclass of `Blog`:

```
class ThemeBlog(Blog):
    theme = models.CharField(max_length=200)

django_blog = ThemeBlog(name='Django', tagline='Django is easy', theme = 'python')
django_blog.save() # django_blog.pk == 3
```

Due to how inheritance works, you have to set both `pk` and `id` to None:

```
django_blog.pk = None
django_blog.id = None
django_blog.save() # django_blog.pk == 4
```

This process does not copy related objects. If you want to copy relations, you have to write a little bit more code. In our example, `Entry` has a many to many field to `Author`:

```
entry = Entry.objects.all()[0] # some previous entry
old_authors = entry.authors.all()
entry.pk = None
entry.save()
entry.authors = old_authors # saves new many2many relations
```

Updating multiple objects at once

Sometimes you want to set a field to a particular value for all the objects in a `QuerySet`. You can do this with the `update()` method. For example:

```
# Update all the headlines with pub_date in 2007.
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

You can only set non-relation fields and `ForeignKey` fields using this method. To update a non-relation field, provide the new value as a constant. To update `ForeignKey` fields, set the new value to be the new model instance you want to point to. For example:

```
>>> b = Blog.objects.get(pk=1)

# Change every Entry so that it belongs to this Blog.
>>> Entry.objects.all().update(blog=b)
```

The `update()` method is applied instantly and returns the number of rows affected by the query. The only restriction on the `QuerySet` that is updated is that it can only access one database table, the model's main table. You can filter based on related fields, but you can only update columns in the model's main table. Example:

```
>>> b = Blog.objects.get(pk=1)

# Update all the headlines belonging to this Blog.
>>> Entry.objects.select_related().filter(blog=b).update(headline='Everything is the same')
```

Be aware that the `update()` method is converted directly to an SQL statement. It is a bulk operation for direct updates. It doesn't run any `save()` methods on your models, or emit the `pre_save` or `post_save` signals (which are a consequence of calling `save()`). If you want to save every item in a `QuerySet` and make sure that the `save()` method is called on each instance, you don't need any special function to handle that. Just loop over them and call `save()`:

```
for item in my_queryset:
    item.save()
```

Calls to update can also use *F() objects* to update one field based on the value of another field in the model. This is especially useful for incrementing counters based upon their current value. For example, to increment the pingback count for every entry in the blog:

```
>>> Entry.objects.all().update(n_pingbacks=F('n_pingbacks') + 1)
```

However, unlike `F()` objects in filter and exclude clauses, you can't introduce joins when you use `F()` objects in an update – you can only reference fields local to the model being updated. If you attempt to introduce a join with an `F()` object, a `FieldError` will be raised:

```
# THIS WILL RAISE A FieldError
>>> Entry.objects.update(headline=F('blog__name'))
```

Related objects

When you define a relationship in a model (i.e., a `ForeignKey`, `OneToOneField`, or `ManyToManyField`), instances of that model will have a convenient API to access the related object(s).

Using the models at the top of this page, for example, an `Entry` object `e` can get its associated `Blog` object by accessing the `blog` attribute: `e.blog`.

(Behind the scenes, this functionality is implemented by Python descriptors. This shouldn't really matter to you, but we point it out here for the curious.)

Django also creates API accessors for the "other" side of the relationship – the link from the related model to the model that defines the relationship. For example, a `Blog` object `b` has access to a list of all related `Entry` objects via the `entry_set` attribute: `b.entry_set.all()`.

All examples in this section use the sample `Blog`, `Author` and `Entry` models defined at the top of this page.

One-to-many relationships

Forward If a model has a `ForeignKey`, instances of that model will have access to the related (foreign) object via a simple attribute of the model.

Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Returns the related Blog object.
```

You can get and set via a foreign-key attribute. As you may expect, changes to the foreign key aren't saved to the database until you call `save()`. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

If a `ForeignKey` field has `null=True` set (i.e., it allows `NULL` values), you can assign `None` to it. Example:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

Forward access to one-to-many relationships is cached the first time the related object is accessed. Subsequent accesses to the foreign key on the same object instance are cached. Example:

```
>>> e = Entry.objects.get(id=2)
>>> print(e.blog)  # Hits the database to retrieve the associated Blog.
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
```

Note that the `select_related()` `QuerySet` method recursively prepopulates the cache of all one-to-many relationships ahead of time. Example:

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
>>> print(e.blog)  # Doesn't hit the database; uses cached version.
```

Following relationships "backward" If a model has a `ForeignKey`, instances of the foreign-key model will have access to a `Manager` that returns all instances of the first model. By default, this `Manager` is named `FOO_set`, where `FOO` is the source model name, lowercased. This `Manager` returns `QuerySets`, which can be filtered and manipulated as described in the "Retrieving objects" section above.

Example:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Returns all Entry objects related to Blog.

# b.entry_set is a Manager that returns QuerySets.
>>> b.entry_set.filter(headline__contains='Lennon')
>>> b.entry_set.count()
```

You can override the `FOO_set` name by setting the `related_name` parameter in the `ForeignKey()` definition. For example, if the `Entry` model was altered to `blog = ForeignKey(Blog, related_name='entries')`, the above example code would look like this:

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Returns all Entry objects related to Blog.

# b.entries is a Manager that returns QuerySets.
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

You cannot access a reverse `ForeignKey` `Manager` from the class; it must be accessed from an instance:

```
>>> Blog.entry_set
Traceback:
    ...
AttributeError: "Manager must be accessed via instance".
```

In addition to the `QuerySet` methods defined in "Retrieving objects" above, the `ForeignKey` `Manager` has additional methods used to handle the set of related objects. A synopsis of each is below, and complete details can be found in the *related objects reference*.

`add(obj1, obj2, ...)` Adds the specified model objects to the related object set.

`create(**kwargs)` Creates a new object, saves it and puts it in the related object set. Returns the newly created object.

remove(obj1, obj2, ...) Removes the specified model objects from the related object set.

clear() Removes all objects from the related object set.

To assign the members of a related set in one fell swoop, just assign to it from any iterable object. The iterable can contain object instances, or just a list of primary key values. For example:

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

In this example, `e1` and `e2` can be full Entry instances, or integer primary key values.

If the `clear()` method is available, any pre-existing objects will be removed from the `entry_set` before all objects in the iterable (in this case, a list) are added to the set. If the `clear()` method is *not* available, all objects in the iterable will be added without removing any existing elements.

Each "reverse" operation described in this section has an immediate effect on the database. Every addition, creation and deletion is immediately and automatically saved to the database.

Many-to-many relationships

Both ends of a many-to-many relationship get automatic API access to the other end. The API works just as a "backward" one-to-many relationship, above.

The only difference is in the attribute naming: The model that defines the `ManyToManyField` uses the attribute name of that field itself, whereas the "reverse" model uses the lowercased model name of the original model, plus `'_set'` (just like reverse one-to-many relationships).

An example makes this easier to understand:

```
e = Entry.objects.get(id=3)
e.authors.all() # Returns all Author objects for this Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Returns all Entry objects for this Author.
```

Like `ForeignKey`, `ManyToManyField` can specify `related_name`. In the above example, if the `ManyToManyField` in `Entry` had specified `related_name='entries'`, then each `Author` instance would have an `entries` attribute instead of `entry_set`.

One-to-one relationships

One-to-one relationships are very similar to many-to-one relationships. If you define a `OneToOneField` on your model, instances of that model will have access to the related object via a simple attribute of the model.

For example:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Returns the related Entry object.
```

The difference comes in "reverse" queries. The related model in a one-to-one relationship also has access to a `Manager` object, but that `Manager` represents a single object, rather than a collection of objects:

```
e = Entry.objects.get(id=2)
e.entrydetail # returns the related EntryDetail object
```

If no object has been assigned to this relationship, Django will raise a `DoesNotExist` exception.

Instances can be assigned to the reverse relationship in the same way as you would assign the forward relationship:

```
e.entrydetail = ed
```

How are the backward relationships possible?

Other object-relational mappers require you to define relationships on both sides. The Django developers believe this is a violation of the DRY (Don't Repeat Yourself) principle, so Django only requires you to define the relationship on one end.

But how is this possible, given that a model class doesn't know which other model classes are related to it until those other model classes are loaded?

The answer lies in the `INSTALLED_APPS` setting. The first time any model is loaded, Django iterates over every model in `INSTALLED_APPS` and creates the backward relationships in memory as needed. Essentially, one of the functions of `INSTALLED_APPS` is to tell Django the entire model domain.

Queries over related objects

Queries involving related objects follow the same rules as queries involving normal value fields. When specifying the value for a query to match, you may use either an object instance itself, or the primary key value for the object.

For example, if you have a Blog object `b` with `id=5`, the following three queries would be identical:

```
Entry.objects.filter(blog=b) # Query using object instance
Entry.objects.filter(blog=b.id) # Query using id from instance
Entry.objects.filter(blog=5) # Query using id directly
```

Falling back to raw SQL

If you find yourself needing to write an SQL query that is too complex for Django's database-mapper to handle, you can fall back on writing SQL by hand. Django has a couple of options for writing raw SQL queries; see *Performing raw SQL queries*.

Finally, it's important to note that the Django database layer is merely an interface to your database. You can access your database via other tools, programming languages or database frameworks; there's nothing Django-specific about your database.

3.2.3 Aggregation

The topic guide on *Django's database-abstraction API* described the way that you can use Django queries that create, retrieve, update and delete individual objects. However, sometimes you will need to retrieve values that are derived by summarizing or *aggregating* a collection of objects. This topic guide describes the ways that aggregate values can be generated and returned using Django queries.

Throughout this guide, we'll refer to the following models. These models are used to track the inventory for a series of online bookstores:

```
class Author(models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()
    friends = models.ManyToManyField('self', blank=True)

class Publisher(models.Model):
    name = models.CharField(max_length=300)
    num_awards = models.IntegerField()

class Book(models.Model):
    isbn = models.CharField(max_length=9)
    name = models.CharField(max_length=300)
    pages = models.IntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.FloatField()
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
    pubdate = models.DateField()

class Store(models.Model):
    name = models.CharField(max_length=300)
    books = models.ManyToManyField(Book)
```

Cheat sheet

In a hurry? Here's how to do common aggregate queries, assuming the models above:

```
# Total number of books.
>>> Book.objects.count()
2452

# Total number of books with publisher=BaloneyPress
>>> Book.objects.filter(publisher__name='BaloneyPress').count()
73

# Average price across all books.
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}

# Max price across all books.
>>> from django.db.models import Max
>>> Book.objects.all().aggregate(Max('price'))
{'price__max': Decimal('81.20')}

# Each publisher, each with a count of books as a "num_books" attribute.
>>> from django.db.models import Count
>>> pubs = Publisher.objects.annotate(num_books=Count('book'))
>>> pubs
[<Publisher BaloneyPress>, <Publisher SalamiPress>, ...]
>>> pubs[0].num_books
73

# The top 5 publishers, in order by number of books.
>>> from django.db.models import Count
>>> pubs = Publisher.objects.annotate(num_books=Count('book')).order_by('-num_books')[:5]
>>> pubs[0].num_books
```

```
1323
```

Generating aggregates over a QuerySet

Django provides two ways to generate aggregates. The first way is to generate summary values over an entire `QuerySet`. For example, say you wanted to calculate the average price of all books available for sale. Django's query syntax provides a means for describing the set of all books:

```
>>> Book.objects.all()
```

What we need is a way to calculate summary values over the objects that belong to this `QuerySet`. This is done by appending an `aggregate()` clause onto the `QuerySet`:

```
>>> from django.db.models import Avg
>>> Book.objects.all().aggregate(Avg('price'))
{'price__avg': 34.35}
```

The `all()` is redundant in this example, so this could be simplified to:

```
>>> Book.objects.aggregate(Avg('price'))
{'price__avg': 34.35}
```

The argument to the `aggregate()` clause describes the aggregate value that we want to compute - in this case, the average of the `price` field on the `Book` model. A list of the aggregate functions that are available can be found in the *QuerySet reference*.

`aggregate()` is a terminal clause for a `QuerySet` that, when invoked, returns a dictionary of name-value pairs. The name is an identifier for the aggregate value; the value is the computed aggregate. The name is automatically generated from the name of the field and the aggregate function. If you want to manually specify a name for the aggregate value, you can do so by providing that name when you specify the aggregate clause:

```
>>> Book.objects.aggregate(average_price=Avg('price'))
{'average_price': 34.35}
```

If you want to generate more than one aggregate, you just add another argument to the `aggregate()` clause. So, if we also wanted to know the maximum and minimum price of all books, we would issue the query:

```
>>> from django.db.models import Avg, Max, Min, Count
>>> Book.objects.aggregate(Avg('price'), Max('price'), Min('price'))
{'price__avg': 34.35, 'price__max': Decimal('81.20'), 'price__min': Decimal('12.99')}
```

Generating aggregates for each item in a QuerySet

The second way to generate summary values is to generate an independent summary for each object in a `QuerySet`. For example, if you are retrieving a list of books, you may want to know how many authors contributed to each book. Each Book has a many-to-many relationship with the Author; we want to summarize this relationship for each book in the `QuerySet`.

Per-object summaries can be generated using the `annotate()` clause. When an `annotate()` clause is specified, each object in the `QuerySet` will be annotated with the specified values.

The syntax for these annotations is identical to that used for the `aggregate()` clause. Each argument to `annotate()` describes an aggregate that is to be calculated. For example, to annotate Books with the number of authors:

```
# Build an annotated queryset
>>> q = Book.objects.annotate(Count('authors'))
# Interrogate the first object in the queryset
>>> q[0]
<Book: The Definitive Guide to Django>
>>> q[0].authors__count
2
# Interrogate the second object in the queryset
>>> q[1]
<Book: Practical Django Projects>
>>> q[1].authors__count
1
```

As with `aggregate()`, the name for the annotation is automatically derived from the name of the aggregate function and the name of the field being aggregated. You can override this default name by providing an alias when you specify the annotation:

```
>>> q = Book.objects.annotate(num_authors=Count('authors'))
>>> q[0].num_authors
2
>>> q[1].num_authors
1
```

Unlike `aggregate()`, `annotate()` is *not* a terminal clause. The output of the `annotate()` clause is a `QuerySet`; this `QuerySet` can be modified using any other `QuerySet` operation, including `filter()`, `order_by`, or even additional calls to `annotate()`.

Joins and aggregates

So far, we have dealt with aggregates over fields that belong to the model being queried. However, sometimes the value you want to aggregate will belong to a model that is related to the model you are querying.

When specifying the field to be aggregated in an aggregate function, Django will allow you to use the same *double underscore notation* that is used when referring to related fields in filters. Django will then handle any table joins that are required to retrieve and aggregate the related value.

For example, to find the price range of books offered in each store, you could use the annotation:

```
>>> Store.objects.annotate(min_price=Min('books__price'), max_price=Max('books__price'))
```

This tells Django to retrieve the Store model, join (through the many-to-many relationship) with the Book model, and aggregate on the price field of the book model to produce a minimum and maximum value.

The same rules apply to the `aggregate()` clause. If you wanted to know the lowest and highest price of any book that is available for sale in a store, you could use the aggregate:

```
>>> Store.objects.aggregate(min_price=Min('books__price'), max_price=Max('books__price'))
```

Join chains can be as deep as you require. For example, to extract the age of the youngest author of any book available for sale, you could issue the query:

```
>>> Store.objects.aggregate(youngest_age=Min('books__authors__age'))
```

Aggregations and other QuerySet clauses

filter() and exclude()

Aggregates can also participate in filters. Any `filter()` (or `exclude()`) applied to normal model fields will have the effect of constraining the objects that are considered for aggregation.

When used with an `annotate()` clause, a filter has the effect of constraining the objects for which an annotation is calculated. For example, you can generate an annotated list of all books that have a title starting with "Django" using the query:

```
>>> Book.objects.filter(name__startswith="Django").annotate(num_authors=Count('authors'))
```

When used with an `aggregate()` clause, a filter has the effect of constraining the objects over which the aggregate is calculated. For example, you can generate the average price of all books with a title that starts with "Django" using the query:

```
>>> Book.objects.filter(name__startswith="Django").aggregate(Avg('price'))
```

Filtering on annotations Annotated values can also be filtered. The alias for the annotation can be used in `filter()` and `exclude()` clauses in the same way as any other model field.

For example, to generate a list of books that have more than one author, you can issue the query:

```
>>> Book.objects.annotate(num_authors=Count('authors')).filter(num_authors__gt=1)
```

This query generates an annotated result set, and then generates a filter based upon that annotation.

Order of annotate() and filter() clauses When developing a complex query that involves both `annotate()` and `filter()` clauses, particular attention should be paid to the order in which the clauses are applied to the `QuerySet`.

When an `annotate()` clause is applied to a query, the annotation is computed over the state of the query up to the point where the annotation is requested. The practical implication of this is that `filter()` and `annotate()` are not commutative operations – that is, there is a difference between the query:

```
>>> Publisher.objects.annotate(num_books=Count('book')).filter(book__rating__gt=3.0)
```

and the query:

```
>>> Publisher.objects.filter(book__rating__gt=3.0).annotate(num_books=Count('book'))
```

Both queries will return a list of Publishers that have at least one good book (i.e., a book with a rating exceeding 3.0). However, the annotation in the first query will provide the total number of all books published by the publisher; the second query will only include good books in the annotated count. In the first query, the annotation precedes the filter, so the filter has no effect on the annotation. In the second query, the filter precedes the annotation, and as a result, the filter constrains the objects considered when calculating the annotation.

order_by()

Annotations can be used as a basis for ordering. When you define an `order_by()` clause, the aggregates you provide can reference any alias defined as part of an `annotate()` clause in the query.

For example, to order a `QuerySet` of books by the number of authors that have contributed to the book, you could use the following query:

```
>>> Book.objects.annotate(num_authors=Count('authors')).order_by('num_authors')
```

values()

Ordinarily, annotations are generated on a per-object basis - an annotated `QuerySet` will return one result for each object in the original `QuerySet`. However, when a `values()` clause is used to constrain the columns that are returned in the result set, the method for evaluating annotations is slightly different. Instead of returning an annotated result for each result in the original `QuerySet`, the original results are grouped according to the unique combinations of the fields specified in the `values()` clause. An annotation is then provided for each unique group; the annotation is computed over all members of the group.

For example, consider an author query that attempts to find out the average rating of books written by each author:

```
>>> Author.objects.annotate(average_rating=Avg('book__rating'))
```

This will return one result for each author in the database, annotated with their average book rating.

However, the result will be slightly different if you use a `values()` clause:

```
>>> Author.objects.values('name').annotate(average_rating=Avg('book__rating'))
```

In this example, the authors will be grouped by name, so you will only get an annotated result for each *unique* author name. This means if you have two authors with the same name, their results will be merged into a single result in the output of the query; the average will be computed as the average over the books written by both authors.

Order of `annotate()` and `values()` clauses As with the `filter()` clause, the order in which `annotate()` and `values()` clauses are applied to a query is significant. If the `values()` clause precedes the `annotate()`, the annotation will be computed using the grouping described by the `values()` clause.

However, if the `annotate()` clause precedes the `values()` clause, the annotations will be generated over the entire query set. In this case, the `values()` clause only constrains the fields that are generated on output.

For example, if we reverse the order of the `values()` and `annotate()` clause from our previous example:

```
>>> Author.objects.annotate(average_rating=Avg('book__rating')).values('name', 'average_rating')
```

This will now yield one unique result for each author; however, only the author's name and the `average_rating` annotation will be returned in the output data.

You should also note that `average_rating` has been explicitly included in the list of values to be returned. This is required because of the ordering of the `values()` and `annotate()` clause.

If the `values()` clause precedes the `annotate()` clause, any annotations will be automatically added to the result set. However, if the `values()` clause is applied after the `annotate()` clause, you need to explicitly include the aggregate column.

Interaction with default ordering or `order_by()` Fields that are mentioned in the `order_by()` part of a queryset (or which are used in the default ordering on a model) are used when selecting the output data, even if they are not otherwise specified in the `values()` call. These extra fields are used to group "like" results together and they can make otherwise identical result rows appear to be separate. This shows up, particularly, when counting things.

By way of example, suppose you have a model like this:

```
class Item(models.Model):
    name = models.CharField(max_length=10)
    data = models.IntegerField()
```

```
    class Meta:
        ordering = ["name"]
```

The important part here is the default ordering on the `name` field. If you want to count how many times each distinct `data` value appears, you might try this:

```
# Warning: not quite correct!
Item.objects.values("data").annotate(Count("id"))
```

...which will group the `Item` objects by their common `data` values and then count the number of `id` values in each group. Except that it won't quite work. The default ordering by `name` will also play a part in the grouping, so this query will group by distinct `(data, name)` pairs, which isn't what you want. Instead, you should construct this queryset:

```
Item.objects.values("data").annotate(Count("id")).order_by()
```

...clearing any ordering in the query. You could also order by, say, `data` without any harmful effects, since that is already playing a role in the query.

This behavior is the same as that noted in the queryset documentation for `distinct()` and the general rule is the same: normally you won't want extra columns playing a part in the result, so clear out the ordering, or at least make sure it's restricted only to those fields you also select in a `values()` call.

Note: You might reasonably ask why Django doesn't remove the extraneous columns for you. The main reason is consistency with `distinct()` and other places: Django **never** removes ordering constraints that you have specified (and we can't change those other methods' behavior, as that would violate our *API stability* policy).

Aggregating annotations

You can also generate an aggregate on the result of an annotation. When you define an `aggregate()` clause, the aggregates you provide can reference any alias defined as part of an `annotate()` clause in the query.

For example, if you wanted to calculate the average number of authors per book you first annotate the set of books with the author count, then aggregate that author count, referencing the annotation field:

```
>>> Book.objects.annotate(num_authors=Count('authors')).aggregate(Avg('num_authors'))
{'num_authors__avg': 1.66}
```

3.2.4 Managers

class `Manager`

A `Manager` is the interface through which database query operations are provided to Django models. At least one `Manager` exists for every model in a Django application.

The way `Manager` classes work is documented in *Making queries*; this document specifically touches on model options that customize `Manager` behavior.

Manager names

By default, Django adds a `Manager` with the name `objects` to every Django model class. However, if you want to use `objects` as a field name, or if you want to use a name other than `objects` for the `Manager`, you can rename it on a per-model basis. To rename the `Manager` for a given class, define a class attribute of type `models.Manager()` on that model. For example:

```
from django.db import models

class Person(models.Model):
    #...
    people = models.Manager()
```

Using this example model, `Person.objects` will generate an `AttributeError` exception, but `Person.people.all()` will provide a list of all `Person` objects.

Custom Managers

You can use a custom `Manager` in a particular model by extending the base `Manager` class and instantiating your custom `Manager` in your model.

There are two reasons you might want to customize a `Manager`: to add extra `Manager` methods, and/or to modify the initial `QuerySet` the `Manager` returns.

Adding extra Manager methods

Adding extra `Manager` methods is the preferred way to add "table-level" functionality to your models. (For "row-level" functionality – i.e., functions that act on a single instance of a model object – use *Model methods*, not custom `Manager` methods.)

A custom `Manager` method can return anything you want. It doesn't have to return a `QuerySet`.

For example, this custom `Manager` offers a method `with_counts()`, which returns a list of all `OpinionPoll` objects, each with an extra `num_responses` attribute that is the result of an aggregate query:

```
class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY 1, 2, 3
            ORDER BY 3 DESC""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(Poll)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```

With this example, you'd use `OpinionPoll.objects.with_counts()` to return that list of `OpinionPoll` objects with `num_responses` attributes.

Another thing to note about this example is that `Manager` methods can access `self.model` to get the model class to which they're attached.

Modifying initial Manager QuerySets

A `Manager`'s base `QuerySet` returns all objects in the system. For example, using this model:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
```

...the statement `Book.objects.all()` will return all books in the database.

You can override a `Manager`'s base `QuerySet` by overriding the `Manager.get_query_set()` method. `get_query_set()` should return a `QuerySet` with the properties you require.

For example, the following model has *two* `Managers` – one that returns all objects, and one that returns only the books by Roald Dahl:

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_query_set(self):
        return super(DahlBookManager, self).get_query_set().filter(author='Roald Dahl')

# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager.
```

With this sample model, `Book.objects.all()` will return all books in the database, but `Book.dahl_objects.all()` will only return the ones written by Roald Dahl.

Of course, because `get_query_set()` returns a `QuerySet` object, you can use `filter()`, `exclude()` and all the other `QuerySet` methods on it. So these statements are all legal:

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

This example also pointed out another interesting technique: using multiple managers on the same model. You can attach as many `Manager()` instances to a model as you'd like. This is an easy way to define common "filters" for your models.

For example:

```
class MaleManager(models.Manager):
    def get_query_set(self):
        return super(MaleManager, self).get_query_set().filter(sex='M')

class FemaleManager(models.Manager):
    def get_query_set(self):
        return super(FemaleManager, self).get_query_set().filter(sex='F')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
```

```
    sex = models.CharField(max_length=1, choices=(('M', 'Male'), ('F', 'Female')))
    people = models.Manager()
    men = MaleManager()
    women = FemaleManager()
```

This example allows you to request `Person.men.all()`, `Person.women.all()`, and `Person.people.all()`, yielding predictable results.

If you use custom `Manager` objects, take note that the first `Manager` Django encounters (in the order in which they're defined in the model) has a special status. Django interprets the first `Manager` defined in a class as the "default" `Manager`, and several parts of Django (including `dumpdata`) will use that `Manager` exclusively for that model. As a result, it's a good idea to be careful in your choice of default manager in order to avoid a situation where overriding `get_query_set()` results in an inability to retrieve objects you'd like to work with.

Using managers for related object access By default, Django uses an instance of a "plain" manager class when accessing related objects (i.e. `choice.poll`), not the default manager on the related object. This is because Django needs to be able to retrieve the related object, even if it would otherwise be filtered out (and hence be inaccessible) by the default manager.

If the normal plain manager class (`django.db.models.Manager`) is not appropriate for your circumstances, you can force Django to use the same class as the default manager for your model by setting the *use_for_related_fields* attribute on the manager class. This is documented fully below.

Custom managers and model inheritance

Class inheritance and model managers aren't quite a perfect match for each other. Managers are often specific to the classes they are defined on and inheriting them in subclasses isn't necessarily a good idea. Also, because the first manager declared is the *default manager*, it is important to allow that to be controlled. So here's how Django handles custom managers and *model inheritance*:

1. Managers defined on non-abstract base classes are *not* inherited by child classes. If you want to reuse a manager from a non-abstract base, redeclare it explicitly on the child class. These sorts of managers are likely to be fairly specific to the class they are defined on, so inheriting them can often lead to unexpected results (particularly as far as the default manager goes). Therefore, they aren't passed onto child classes.
2. Managers from abstract base classes are always inherited by the child class, using Python's normal name resolution order (names on the child class override all others; then come names on the first parent class, and so on). Abstract base classes are designed to capture information and behavior that is common to their child classes. Defining common managers is an appropriate part of this common information.
3. The default manager on a class is either the first manager declared on the class, if that exists, or the default manager of the first abstract base class in the parent hierarchy, if that exists. If no default manager is explicitly declared, Django's normal default manager is used.

These rules provide the necessary flexibility if you want to install a collection of custom managers on a group of models, via an abstract base class, but still customize the default manager. For example, suppose you have this base class:

```
class AbstractBase(models.Model):
    ...
    objects = CustomManager()

    class Meta:
        abstract = True
```

If you use this directly in a subclass, `objects` will be the default manager if you declare no managers in the base class:

```
class ChildA(AbstractBase):
    ...
    # This class has CustomManager as the default manager.
```

If you want to inherit from `AbstractBase`, but provide a different default manager, you can provide the default manager on the child class:

```
class ChildB(AbstractBase):
    ...
    # An explicit default manager.
    default_manager = OtherManager()
```

Here, `default_manager` is the default. The `objects` manager is still available, since it's inherited. It just isn't used as the default.

Finally for this example, suppose you want to add extra managers to the child class, but still use the default from `AbstractBase`. You can't add the new manager directly in the child class, as that would override the default and you would have to also explicitly include all the managers from the abstract base class. The solution is to put the extra managers in another base class and introduce it into the inheritance hierarchy *after* the defaults:

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()

    class Meta:
        abstract = True

class ChildC(AbstractBase, ExtraManager):
    ...
    # Default manager is CustomManager, but OtherManager is
    # also available via the "extra_manager" attribute.
```

Implementation concerns

Whatever features you add to your custom `Manager`, it must be possible to make a shallow copy of a `Manager` instance; i.e., the following code must work:

```
>>> import copy
>>> manager = MyManager()
>>> my_copy = copy.copy(manager)
```

Django makes shallow copies of manager objects during certain queries; if your Manager cannot be copied, those queries will fail.

This won't be an issue for most custom managers. If you are just adding simple methods to your `Manager`, it is unlikely that you will inadvertently make instances of your `Manager` uncopyable. However, if you're overriding `__getattr__` or some other private method of your `Manager` object that controls object state, you should ensure that you don't affect the ability of your `Manager` to be copied.

Controlling automatic Manager types

This document has already mentioned a couple of places where Django creates a manager class for you: default managers and the "plain" manager used to access related objects. There are other places in the implementation of Django where temporary plain managers are needed. Those automatically created managers will normally be instances of the `django.db.models.Manager` class. Throughout this section, we will use the term "automatic manager" to mean a manager that Django creates for you – either as a default manager on a model with no managers, or to use temporarily when accessing related objects.

Sometimes this default class won't be the right choice. One example is in the `django.contrib.gis` application that ships with Django itself. All `gis` models must use a special manager class (`GeoManager`) because they need a special queryset (`GeoQuerySet`) to be used for interacting with the database. It turns out that models which require a special manager like this need to use the same manager class wherever an automatic manager is created.

Django provides a way for custom manager developers to say that their manager class should be used for automatic managers whenever it is the default manager on a model. This is done by setting the `use_for_related_fields` attribute on the manager class:

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

    ...
```

If this attribute is set on the *default* manager for a model (only the default manager is considered in these situations), Django will use that class whenever it needs to automatically create a manager for the class. Otherwise, it will use `django.db.models.Manager`.

Historical Note

Given the purpose for which it's used, the name of this attribute (`use_for_related_fields`) might seem a little odd. Originally, the attribute only controlled the type of manager used for related field access, which is where the name came from. As it became clear the concept was more broadly useful, the name hasn't been changed. This is primarily so that existing code will *continue to work* in future Django versions.

Writing correct Managers for use in automatic Manager instances

As already suggested by the *django.contrib.gis* example, above, the `use_for_related_fields` feature is primarily for managers that need to return a custom `QuerySet` subclass. In providing this functionality in your manager, there are a couple of things to remember.

Do not filter away any results in this type of manager subclass One reason an automatic manager is used is to access objects that are related to from some other model. In those situations, Django has to be able to see all the objects for the model it is fetching, so that *anything* which is referred to can be retrieved.

If you override the `get_query_set()` method and filter out any rows, Django will return incorrect results. Don't do that. A manager that filters results in `get_query_set()` is not appropriate for use as an automatic manager.

Set `use_for_related_fields` when you define the class The `use_for_related_fields` attribute must be set on the manager *class*, not on an *instance* of the class. The earlier example shows the correct way to set it, whereas the following will not work:

```
# BAD: Incorrect code
class MyManager(models.Manager):
    ...

# Sets the attribute on an instance of MyManager. Django will
# ignore this setting.
mgr = MyManager()
mgr.use_for_related_fields = True

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

```
# End of incorrect code.
```

You also shouldn't change the attribute on the class object after it has been used in a model, since the attribute's value is processed when the model class is created and not subsequently reread. Set the attribute on the manager class when it is first defined, as in the initial example of this section and everything will work smoothly.

3.2.5 Performing raw SQL queries

When the *model query APIs* don't go far enough, you can fall back to writing raw SQL. Django gives you two ways of performing raw SQL queries: you can use `Manager.raw()` to perform raw queries and return model instances, or you can avoid the model layer entirely and execute custom SQL directly.

Performing raw queries

The `raw()` manager method can be used to perform raw SQL queries that return model instances:

`Manager.`**`raw`**(*raw_query*, *params=None*, *translations=None*)

This method method takes a raw SQL query, executes it, and returns a `RawQuerySet` instance. This `RawQuerySet` instance can be iterated over just like an normal QuerySet to provide object instances.

This is best illustrated with an example. Suppose you've got the following model:

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

You could then execute custom SQL like so:

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

Of course, this example isn't very exciting – it's exactly the same as running `Person.objects.all()`. However, `raw()` has a bunch of other options that make it very powerful.

Model table names

Where'd the name of the `Person` table come from in that example?

By default, Django figures out a database table name by joining the model's "app label" – the name you used in `manage.py startapp` – to the model's class name, with an underscore between them. In the example we've assumed that the `Person` model lives in an app named `myapp`, so its table would be `myapp_person`.

For more details check out the documentation for the `db_table` option, which also lets you manually set the database table name.

Warning: No checking is done on the SQL statement that is passed in to `.raw()`. Django expects that the statement will return a set of rows from the database, but does nothing to enforce that. If the query does not return rows, a (possibly cryptic) error will result.

Mapping query fields to model fields

`raw()` automatically maps fields in the query to fields on the model.

The order of fields in your query doesn't matter. In other words, both of the following queries work identically:

```
>>> Person.objects.raw('SELECT id, first_name, last_name, birth_date FROM myapp_person')
...
>>> Person.objects.raw('SELECT last_name, birth_date, first_name, id FROM myapp_person')
...
```

Matching is done by name. This means that you can use SQL's `AS` clauses to map fields in the query to model fields. So if you had some other table that had `Person` data in it, you could easily map it into `Person` instances:

```
>>> Person.objects.raw('''SELECT first AS first_name,
...                              last AS last_name,
...                              bd AS birth_date,
...                              pk as id,
...                       FROM some_other_table''')
```

As long as the names match, the model instances will be created correctly.

Alternatively, you can map fields in the query to model fields using the `translations` argument to `raw()`. This is a dictionary mapping names of fields in the query to names of fields on the model. For example, the above query could also be written:

```
>>> name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date', 'pk': 'id'}
>>> Person.objects.raw('SELECT * FROM some_other_table', translations=name_map)
```

Index lookups

`raw()` supports indexing, so if you need only the first result you can write:

```
>>> first_person = Person.objects.raw('SELECT * from myapp_person')[0]
```

However, the indexing and slicing are not performed at the database level. If you have a big amount of `Person` objects in your database, it is more efficient to limit the query at the SQL level:

```
>>> first_person = Person.objects.raw('SELECT * from myapp_person LIMIT 1')[0]
```

Deferring model fields

Fields may also be left out:

```
>>> people = Person.objects.raw('SELECT id, first_name FROM myapp_person')
```

The `Person` objects returned by this query will be deferred model instances (see `defer()`). This means that the fields that are omitted from the query will be loaded on demand. For example:

```
>>> for p in Person.objects.raw('SELECT id, first_name FROM myapp_person'):
...     print(p.first_name, # This will be retrieved by the original query
...           p.last_name) # This will be retrieved on demand
...
John Smith
Jane Jones
```

From outward appearances, this looks like the query has retrieved both the first name and last name. However, this example actually issued 3 queries. Only the first names were retrieved by the raw() query – the last names were both retrieved on demand when they were printed.

There is only one field that you can't leave out - the primary key field. Django uses the primary key to identify model instances, so it must always be included in a raw query. An `InvalidQuery` exception will be raised if you forget to include the primary key.

Adding annotations

You can also execute queries containing fields that aren't defined on the model. For example, we could use PostgreSQL's age() function to get a list of people with their ages calculated by the database:

```
>>> people = Person.objects.raw('SELECT *, age(birth_date) AS age FROM myapp_person')
>>> for p in people:
...     print("%s is %s." % (p.first_name, p.age))
John is 37.
Jane is 42.
...
```

Passing parameters into `raw()`

If you need to perform parameterized queries, you can use the `params` argument to `raw()`:

```
>>> lname = 'Doe'
>>> Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s', [lname])
```

`params` is a list of parameters. You'll use `%s` placeholders in the query string (regardless of your database engine); they'll be replaced with parameters from the `params` list.

Warning: Do not use string formatting on raw queries!
It's tempting to write the above query as:

```
>>> query = 'SELECT * FROM myapp_person WHERE last_name = %s' % lname
>>> Person.objects.raw(query)
```

Don't.
Using the `params` list completely protects you from SQL injection attacks, a common exploit where attackers inject arbitrary SQL into your database. If you use string interpolation, sooner or later you'll fall victim to SQL injection. As long as you remember to always use the `params` list you'll be protected.

Executing custom SQL directly

Sometimes even `Manager.raw()` isn't quite enough: you might need to perform queries that don't map cleanly to models, or directly execute `UPDATE`, `INSERT`, or `DELETE` queries.

In these cases, you can always access the database directly, routing around the model layer entirely.

The object `django.db.connection` represents the default database connection, and `django.db.transaction` represents the default database transaction. To use the database connection, call `connection.cursor()` to get a cursor object. Then, call `cursor.execute(sql, [params])` to execute the SQL and `cursor.fetchone()` or `cursor.fetchall()` to return the resulting rows. After performing a data changing operation, you should then call `transaction.commit_unless_managed()` to

ensure your changes are committed to the database. If your query is purely a data retrieval operation, no commit is required. For example:

```
def my_custom_sql():
    from django.db import connection, transaction
    cursor = connection.cursor()

    # Data modifying operation - commit required
    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
    transaction.commit_unless_managed()

    # Data retrieval operation - no commit required
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()

    return row
```

If you are using *more than one database*, you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
# Your code here...
transaction.commit_unless_managed(using='my_db_alias')
```

By default, the Python DB API will return results without their field names, which means you end up with a `list` of values, rather than a `dict`. At a small performance cost, you can return results as a `dict` by using something like this:

```
def dictfetchall(cursor):
    "Returns all rows from a cursor as a dict"
    desc = cursor.description
    return [
        dict(zip([col[0] for col in desc], row))
        for row in cursor.fetchall()
    ]
```

Here is an example of the difference between the two:

```
>>> cursor.execute("SELECT id, parent_id from test LIMIT 2");
>>> cursor.fetchall()
((54360982L, None), (54360880L, None))

>>> cursor.execute("SELECT id, parent_id from test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982L}, {'parent_id': None, 'id': 54360880L}]
```

Transactions and raw SQL

When you make a raw SQL call, Django will automatically mark the current transaction as dirty. You must then ensure that the transaction containing those calls is closed correctly. See *the notes on the requirements of Django's transaction handling* for more details. Changed in version 1.3: *Please see the release notes* Prior to Django 1.3, it was necessary to manually mark a transaction as dirty using `transaction.set_dirty()` when using raw SQL calls.

Connections and cursors

`connection` and `cursor` mostly implement the standard Python DB-API described in **PEP 249** (except when it comes to *transaction handling*). If you're not familiar with the Python DB-API, note that the SQL statement in `cursor.execute()` uses placeholders, `"%s"`, rather than adding parameters directly within the SQL. If you use this technique, the underlying database library will automatically add quotes and escaping to your parameter(s) as necessary. (Also note that Django expects the `"%s"` placeholder, *not* the `"?"` placeholder, which is used by the SQLite Python bindings. This is for the sake of consistency and sanity.)

3.2.6 Managing database transactions

Django gives you a few ways to control how database transactions are managed, if you're using a database that supports transactions.

Django's default transaction behavior

Django's default behavior is to run with an open transaction which it commits automatically when any built-in, data-altering model function is called. For example, if you call `model.save()` or `model.delete()`, the change will be committed immediately.

This is much like the auto-commit setting for most databases. As soon as you perform an action that needs to write to the database, Django produces the `INSERT`/`UPDATE`/`DELETE` statements and then does the `COMMIT`. There's no implicit `ROLLBACK`.

Tying transactions to HTTP requests

The recommended way to handle transactions in Web requests is to tie them to the request and response phases via Django's `TransactionMiddleware`.

It works like this: When a request starts, Django starts a transaction. If the response is produced without problems, Django commits any pending transactions. If the view function produces an exception, Django rolls back any pending transactions.

To activate this feature, just add the `TransactionMiddleware` middleware to your `MIDDLEWARE_CLASSES` setting:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.transaction.TransactionMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

The order is quite important. The transaction middleware applies not only to view functions, but also for all middleware modules that come after it. So if you use the session middleware after the transaction middleware, session creation will be part of the transaction.

The various cache middlewares are an exception: `CacheMiddleware`, `UpdateCacheMiddleware`, and `FetchFromCacheMiddleware` are never affected. Even when using database caching, Django's cache backend uses its own database cursor (which is mapped to its own database connection internally).

Note: The `TransactionMiddleware` only affects the database aliased as "default" within your `DATABASES` setting. If you are using multiple databases and want transaction control over databases other than "default", you will need to write your own transaction middleware.

Controlling transaction management in views

Changed in version 1.3: Transaction management context managers are new in Django 1.3. For most people, implicit request-based transactions work wonderfully. However, if you need more fine-grained control over how transactions are managed, you can use a set of functions in `django.db.transaction` to control transactions on a per-function or per-code-block basis.

These functions, described in detail below, can be used in two different ways:

- As a decorator on a particular function. For example:

```
from django.db import transaction

@transaction.commit_on_success
def viewfunc(request):
    # ...
    # this code executes inside a transaction
    # ...
```

- As a context manager around a particular block of code:

```
from django.db import transaction

def viewfunc(request):
    # ...
    # this code executes using default transaction management
    # ...

    with transaction.commit_on_success():
        # ...
        # this code executes inside a transaction
        # ...
```

Both techniques work with all supported version of Python.

For maximum compatibility, all of the examples below show transactions using the decorator syntax, but all of the follow functions may be used as context managers, too.

Note: Although the examples below use view functions as examples, these decorators and context managers can be used anywhere in your code that you need to deal with transactions.

autocommit()

Use the `autocommit` decorator to switch a view function to Django's default commit behavior, regardless of the global transaction setting.

Example:

```
from django.db import transaction

@transaction.autocommit
def viewfunc(request):
    ....
```

```
@transaction.autocommit(using="my_other_database")
def viewfunc2(request):
    ....
```

Within `viewfunc()`, transactions will be committed as soon as you call `model.save()`, `model.delete()`, or any other function that writes to the database. `viewfunc2()` will have this same behavior, but for the `"my_other_database"` connection.

commit_on_success()

Use the `commit_on_success` decorator to use a single transaction for all the work done in a function:

```
from django.db import transaction

@transaction.commit_on_success
def viewfunc(request):
    ....

@transaction.commit_on_success(using="my_other_database")
def viewfunc2(request):
    ....
```

If the function returns successfully, then Django will commit all work done within the function at that point. If the function raises an exception, though, Django will roll back the transaction.

commit_manually()

Use the `commit_manually` decorator if you need full control over transactions. It tells Django you'll be managing the transaction on your own.

If your view changes data and doesn't `commit()` or `rollback()`, Django will raise a `TransactionManagementError` exception.

Manual transaction management looks like this:

```
from django.db import transaction

@transaction.commit_manually
def viewfunc(request):
    ...
    # You can commit/rollback however and whenever you want
    transaction.commit()
    ...

    # But you've got to remember to do it yourself!
    try:
        ...
    except:
        transaction.rollback()
    else:
        transaction.commit()

@transaction.commit_manually(using="my_other_database")
def viewfunc2(request):
    ....
```

Requirements for transaction handling

New in version 1.3: *Please see the release notes* Django requires that every transaction that is opened is closed before the completion of a request. If you are using `autocommit()` (the default commit mode) or

commit_on_success(), this will be done for you automatically. However, if you are manually managing transactions (using the commit_manually() decorator), you must ensure that the transaction is either committed or rolled back before a request is completed.

This applies to all database operations, not just write operations. Even if your transaction only reads from the database, the transaction must be committed or rolled back before you complete a request.

How to globally deactivate transaction management

Control freaks can totally disable all transaction management by setting DISABLE_TRANSACTION_MANAGEMENT to True in the Django settings file.

If you do this, Django won't provide any automatic transaction management whatsoever. Middleware will no longer implicitly commit transactions, and you'll need to roll management yourself. This even requires you to commit changes done by middleware somewhere else.

Thus, this is best used in situations where you want to run your own transaction-controlling middleware or do something really strange. In almost all situations, you'll be better off using the default behavior, or the transaction middleware, and only modify selected functions as needed.

Savepoints

A savepoint is a marker within a transaction that enables you to roll back part of a transaction, rather than the full transaction. Savepoints are available with the PostgreSQL 8, Oracle and MySQL (when using the InnoDB storage engine) backends. Other backends provide the savepoint functions, but they're empty operations – they don't actually do anything. Changed in version 1.4: Savepoint support for the MySQL backend was added in Django 1.4. Savepoints aren't especially useful if you are using the default autocommit behavior of Django. However, if you are using commit_on_success or commit_manually, each open transaction will build up a series of database operations, awaiting a commit or rollback. If you issue a rollback, the entire transaction is rolled back. Savepoints provide the ability to perform a fine-grained rollback, rather than the full rollback that would be performed by transaction.rollback().

Each of these functions takes a using argument which should be the name of a database for which the behavior applies. If no using argument is provided then the "default" database is used.

Savepoints are controlled by three methods on the transaction object:

transaction.**savepoint**(*using=None*)
: Creates a new savepoint. This marks a point in the transaction that is known to be in a "good" state.

  Returns the savepoint ID (sid).

transaction.**savepoint_commit**(*sid*, *using=None*)
: Updates the savepoint to include any operations that have been performed since the savepoint was created, or since the last commit.

transaction.**savepoint_rollback**(*sid*, *using=None*)
: Rolls the transaction back to the last point at which the savepoint was committed.

The following example demonstrates the use of savepoints:

```
from django.db import transaction

@transaction.commit_manually
def viewfunc(request):

  a.save()
  # open transaction now contains a.save()
```

```
    sid = transaction.savepoint()

    b.save()
    # open transaction now contains a.save() and b.save()

    if want_to_keep_b:
        transaction.savepoint_commit(sid)
        # open transaction still contains a.save() and b.save()
    else:
        transaction.savepoint_rollback(sid)
        # open transaction now contains only a.save()

    transaction.commit()
```

Transactions in MySQL

If you're using MySQL, your tables may or may not support transactions; it depends on your MySQL version and the table types you're using. (By "table types," we mean something like "InnoDB" or "MyISAM".) MySQL transaction peculiarities are outside the scope of this article, but the MySQL site has information on MySQL transactions.

If your MySQL setup does *not* support transactions, then Django will function in auto-commit mode: Statements will be executed and committed as soon as they're called. If your MySQL setup *does* support transactions, Django will handle transactions as explained in this document.

Handling exceptions within PostgreSQL transactions

When a call to a PostgreSQL cursor raises an exception (typically `IntegrityError`), all subsequent SQL in the same transaction will fail with the error "current transaction is aborted, queries ignored until end of transaction block". Whilst simple use of `save()` is unlikely to raise an exception in PostgreSQL, there are more advanced usage patterns which might, such as saving objects with unique fields, saving using the force_insert/force_update flag, or invoking custom SQL.

There are several ways to recover from this sort of error.

Transaction rollback

The first option is to roll back the entire transaction. For example:

```
a.save() # Succeeds, but may be undone by transaction rollback
try:
    b.save() # Could throw exception
except IntegrityError:
    transaction.rollback()
c.save() # Succeeds, but a.save() may have been undone
```

Calling `transaction.rollback()` rolls back the entire transaction. Any uncommitted database operations will be lost. In this example, the changes made by `a.save()` would be lost, even though that operation raised no error itself.

Savepoint rollback

If you are using PostgreSQL 8 or later, you can use *savepoints* to control the extent of a rollback. Before performing a database operation that could fail, you can set or update the savepoint; that way, if the operation fails, you can roll

back the single offending operation, rather than the entire transaction. For example:

```
a.save() # Succeeds, and never undone by savepoint rollback
try:
    sid = transaction.savepoint()
    b.save() # Could throw exception
    transaction.savepoint_commit(sid)
except IntegrityError:
    transaction.savepoint_rollback(sid)
c.save() # Succeeds, and a.save() is never undone
```

In this example, `a.save()` will not be undone in the case where `b.save()` raises an exception.

Database-level autocommit

With PostgreSQL 8.2 or later, there is an advanced option to run PostgreSQL with *database-level autocommit*. If you use this option, there is no constantly open transaction, so it is always possible to continue after catching an exception. For example:

```
a.save() # succeeds
try:
    b.save() # Could throw exception
except IntegrityError:
    pass
c.save() # succeeds
```

Note: This is not the same as the *autocommit decorator*. When using database level autocommit there is no database transaction at all. The `autocommit` decorator still uses transactions, automatically committing each transaction when a database modifying operation occurs.

3.2.7 Multiple databases

This topic guide describes Django's support for interacting with multiple databases. Most of the rest of Django's documentation assumes you are interacting with a single database. If you want to interact with multiple databases, you'll need to take some additional steps.

Defining your databases

The first step to using more than one database with Django is to tell Django about the database servers you'll be using. This is done using the `DATABASES` setting. This setting maps database aliases, which are a way to refer to a specific database throughout Django, to a dictionary of settings for that specific connection. The settings in the inner dictionaries are described fully in the `DATABASES` documentation.

Databases can have any alias you choose. However, the alias `default` has special significance. Django uses the database with the alias of `default` when no other database has been selected. If you don't have a `default` database, you need to be careful to always specify the database that you want to use.

The following is an example `settings.py` snippet defining two databases – a default PostgreSQL database and a MySQL database called `users`:

```
DATABASES = {
    'default': {
        'NAME': 'app_data',
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
```

```
        'USER': 'postgres_user',
        'PASSWORD': 's3krit'
    },
    'users': {
        'NAME': 'user_data',
        'ENGINE': 'django.db.backends.mysql',
        'USER': 'mysql_user',
        'PASSWORD': 'priv4te'
    }
}
```

If you attempt to access a database that you haven't defined in your `DATABASES` setting, Django will raise a `django.db.utils.ConnectionDoesNotExist` exception.

Synchronizing your databases

The `syncdb` management command operates on one database at a time. By default, it operates on the `default` database, but by providing a `--database` argument, you can tell syncdb to synchronize a different database. So, to synchronize all models onto all databases in our example, you would need to call:

```
$ ./manage.py syncdb
$ ./manage.py syncdb --database=users
```

If you don't want every application to be synchronized onto a particular database, you can define a *database router* that implements a policy constraining the availability of particular models.

Alternatively, if you want fine-grained control of synchronization, you can pipe all or part of the output of `sqlall` for a particular application directly into your database prompt, like this:

```
$ ./manage.py sqlall sales | ./manage.py dbshell
```

Using other management commands

The other `django-admin.py` commands that interact with the database operate in the same way as `syncdb` – they only ever operate on one database at a time, using `--database` to control the database used.

Automatic database routing

The easiest way to use multiple databases is to set up a database routing scheme. The default routing scheme ensures that objects remain 'sticky' to their original database (i.e., an object retrieved from the `foo` database will be saved on the same database). The default routing scheme ensures that if a database isn't specified, all queries fall back to the `default` database.

You don't have to do anything to activate the default routing scheme – it is provided 'out of the box' on every Django project. However, if you want to implement more interesting database allocation behaviors, you can define and install your own database routers.

Database routers

A database Router is a class that provides up to four methods:

db_for_read(*model*, ***hints*)
Suggest the database that should be used for read operations for objects of type `model`.

If a database operation is able to provide any additional information that might assist in selecting a database, it will be provided in the `hints` dictionary. Details on valid hints are provided *below*.

Returns None if there is no suggestion.

db_for_write(*model*, ***hints*)
Suggest the database that should be used for writes of objects of type Model.

If a database operation is able to provide any additional information that might assist in selecting a database, it will be provided in the `hints` dictionary. Details on valid hints are provided *below*.

Returns None if there is no suggestion.

allow_relation(*obj1*, *obj2*, ***hints*)
Return True if a relation between obj1 and obj2 should be allowed, False if the relation should be prevented, or None if the router has no opinion. This is purely a validation operation, used by foreign key and many to many operations to determine if a relation should be allowed between two objects.

allow_syncdb(*db*, *model*)
Determine if the `model` should be synchronized onto the database with alias `db`. Return True if the model should be synchronized, False if it should not be synchronized, or None if the router has no opinion. This method can be used to determine the availability of a model on a given database.

A router doesn't have to provide *all* these methods – it may omit one or more of them. If one of the methods is omitted, Django will skip that router when performing the relevant check.

Hints The hints received by the database router can be used to decide which database should receive a given request.

At present, the only hint that will be provided is `instance`, an object instance that is related to the read or write operation that is underway. This might be the instance that is being saved, or it might be an instance that is being added in a many-to-many relation. In some cases, no instance hint will be provided at all. The router checks for the existence of an instance hint, and determine if that hint should be used to alter routing behavior.

Using routers

Database routers are installed using the `DATABASE_ROUTERS` setting. This setting defines a list of class names, each specifying a router that should be used by the master router (`django.db.router`).

The master router is used by Django's database operations to allocate database usage. Whenever a query needs to know which database to use, it calls the master router, providing a model and a hint (if available). Django then tries each router in turn until a database suggestion can be found. If no suggestion can be found, it tries the current `_state.db` of the hint instance. If a hint instance wasn't provided, or the instance doesn't currently have database state, the master router will allocate the `default` database.

An example

Example purposes only!

This example is intended as a demonstration of how the router infrastructure can be used to alter database usage. It intentionally ignores some complex issues in order to demonstrate how routers are used.

This example won't work if any of the models in `myapp` contain relationships to models outside of the `other` database. *Cross-database relationships* introduce referential integrity problems that Django can't currently handle.

The master/slave configuration described is also flawed – it doesn't provide any solution for handling replication lag (i.e., query inconsistencies introduced because of the time taken for a write to propagate to the slaves). It also doesn't consider the interaction of transactions with the database utilization strategy.

So - what does this mean in practice? Say you want `myapp` to exist on the `other` database, and you want all other models in a master/slave relationship between the databases `master`, `slave1` and `slave2`. To implement this, you would need 2 routers:

```
class MyAppRouter(object):
    """A router to control all database operations on models in
    the myapp application"""

    def db_for_read(self, model, **hints):
        "Point all operations on myapp models to 'other'"
        if model._meta.app_label == 'myapp':
            return 'other'
        return None

    def db_for_write(self, model, **hints):
        "Point all operations on myapp models to 'other'"
        if model._meta.app_label == 'myapp':
            return 'other'
        return None

    def allow_relation(self, obj1, obj2, **hints):
        "Allow any relation if a model in myapp is involved"
        if obj1._meta.app_label == 'myapp' or obj2._meta.app_label == 'myapp':
            return True
        return None

    def allow_syncdb(self, db, model):
        "Make sure the myapp app only appears on the 'other' db"
        if db == 'other':
            return model._meta.app_label == 'myapp'
        elif model._meta.app_label == 'myapp':
            return False
        return None

class MasterSlaveRouter(object):
    """A router that sets up a simple master/slave configuration"""

    def db_for_read(self, model, **hints):
        "Point all read operations to a random slave"
        return random.choice(['slave1','slave2'])

    def db_for_write(self, model, **hints):
        "Point all write operations to the master"
        return 'master'

    def allow_relation(self, obj1, obj2, **hints):
        "Allow any relation between two objects in the db pool"
        db_list = ('master','slave1','slave2')
        if obj1._state.db in db_list and obj2._state.db in db_list:
            return True
        return None

    def allow_syncdb(self, db, model):
        "Explicitly put all models on all databases."
        return True
```

Then, in your settings file, add the following (substituting `path.to.` with the actual python path to the module

where you define the routers):

```
DATABASE_ROUTERS = ['path.to.MyAppRouter', 'path.to.MasterSlaveRouter']
```

The order in which routers are processed is significant. Routers will be queried in the order the are listed in the DATABASE_ROUTERS setting . In this example, the MyAppRouter is processed before the MasterSlaveRouter, and as a result, decisions concerning the models in myapp are processed before any other decision is made. If the DATABASE_ROUTERS setting listed the two routers in the other order, MasterSlaveRouter.allow_syncdb() would be processed first. The catch-all nature of the MasterSlaveRouter implementation would mean that all models would be available on all databases.

With this setup installed, lets run some Django code:

```
>>> # This retrieval will be performed on the 'credentials' database
>>> fred = User.objects.get(username='fred')
>>> fred.first_name = 'Frederick'

>>> # This save will also be directed to 'credentials'
>>> fred.save()

>>> # These retrieval will be randomly allocated to a slave database
>>> dna = Person.objects.get(name='Douglas Adams')

>>> # A new object has no database allocation when created
>>> mh = Book(title='Mostly Harmless')

>>> # This assignment will consult the router, and set mh onto
>>> # the same database as the author object
>>> mh.author = dna

>>> # This save will force the 'mh' instance onto the master database...
>>> mh.save()

>>> # ... but if we re-retrieve the object, it will come back on a slave
>>> mh = Book.objects.get(title='Mostly Harmless')
```

Manually selecting a database

Django also provides an API that allows you to maintain complete control over database usage in your code. A manually specified database allocation will take priority over a database allocated by a router.

Manually selecting a database for a QuerySet

You can select the database for a QuerySet at any point in the QuerySet "chain." Just call using() on the QuerySet to get another QuerySet that uses the specified database.

using() takes a single argument: the alias of the database on which you want to run the query. For example:

```
>>> # This will run on the 'default' database.
>>> Author.objects.all()

>>> # So will this.
>>> Author.objects.using('default').all()

>>> # This will run on the 'other' database.
>>> Author.objects.using('other').all()
```

Selecting a database for `save()`

Use the `using` keyword to `Model.save()` to specify to which database the data should be saved.

For example, to save an object to the `legacy_users` database, you'd use this:

```
>>> my_object.save(using='legacy_users')
```

If you don't specify `using`, the `save()` method will save into the default database allocated by the routers.

Moving an object from one database to another If you've saved an instance to one database, it might be tempting to use `save(using=...)` as a way to migrate the instance to a new database. However, if you don't take appropriate steps, this could have some unexpected consequences.

Consider the following example:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')  # (statement 1)
>>> p.save(using='second') # (statement 2)
```

In statement 1, a new `Person` object is saved to the `first` database. At this time, `p` doesn't have a primary key, so Django issues a SQL `INSERT` statement. This creates a primary key, and Django assigns that primary key to `p`.

When the save occurs in statement 2, `p` already has a primary key value, and Django will attempt to use that primary key on the new database. If the primary key value isn't in use in the `second` database, then you won't have any problems – the object will be copied to the new database.

However, if the primary key of `p` is already in use on the `second` database, the existing object in the `second` database will be overridden when `p` is saved.

You can avoid this in two ways. First, you can clear the primary key of the instance. If an object has no primary key, Django will treat it as a new object, avoiding any loss of data on the `second` database:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.pk = None # Clear the primary key.
>>> p.save(using='second') # Write a completely new object.
```

The second option is to use the `force_insert` option to `save()` to ensure that Django does a SQL `INSERT`:

```
>>> p = Person(name='Fred')
>>> p.save(using='first')
>>> p.save(using='second', force_insert=True)
```

This will ensure that the person named `Fred` will have the same primary key on both databases. If that primary key is already in use when you try to save onto the `second` database, an error will be raised.

Selecting a database to delete from

By default, a call to delete an existing object will be executed on the same database that was used to retrieve the object in the first place:

```
>>> u = User.objects.using('legacy_users').get(username='fred')
>>> u.delete() # will delete from the 'legacy_users' database
```

To specify the database from which a model will be deleted, pass a `using` keyword argument to the `Model.delete()` method. This argument works just like the `using` keyword argument to `save()`.

For example, if you're migrating a user from the `legacy_users` database to the `new_users` database, you might use these commands:

```
>>> user_obj.save(using='new_users')
>>> user_obj.delete(using='legacy_users')
```

Using managers with multiple databases

Use the `db_manager()` method on managers to give managers access to a non-default database.

For example, say you have a custom manager method that touches the database – `User.objects.create_user()`. Because `create_user()` is a manager method, not a `QuerySet` method, you can't do `User.objects.using('new_users').create_user()`. (The `create_user()` method is only available on `User.objects`, the manager, not on `QuerySet` objects derived from the manager.) The solution is to use `db_manager()`, like this:

```
User.objects.db_manager('new_users').create_user(...)
```

`db_manager()` returns a copy of the manager bound to the database you specify.

Using `get_query_set()` with multiple databases If you're overriding `get_query_set()` on your manager, be sure to either call the method on the parent (using `super()`) or do the appropriate handling of the `_db` attribute on the manager (a string containing the name of the database to use).

For example, if you want to return a custom `QuerySet` class from the `get_query_set` method, you could do this:

```
class MyManager(models.Manager):
    def get_query_set(self):
        qs = CustomQuerySet(self.model)
        if self._db is not None:
            qs = qs.using(self._db)
        return qs
```

Exposing multiple databases in Django's admin interface

Django's admin doesn't have any explicit support for multiple databases. If you want to provide an admin interface for a model on a database other than that specified by your router chain, you'll need to write custom `ModelAdmin` classes that will direct the admin to use a specific database for content.

`ModelAdmin` objects have five methods that require customization for multiple-database support:

```
class MultiDBModelAdmin(admin.ModelAdmin):
    # A handy constant for the name of the alternate database.
    using = 'other'

    def save_model(self, request, obj, form, change):
        # Tell Django to save objects to the 'other' database.
        obj.save(using=self.using)

    def delete_model(self, request, obj):
        # Tell Django to delete objects from the 'other' database
        obj.delete(using=self.using)

    def queryset(self, request):
        # Tell Django to look for objects on the 'other' database.
        return super(MultiDBModelAdmin, self).queryset(request).using(self.using)
```

```
    def formfield_for_foreignkey(self, db_field, request=None, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_foreignkey(db_field, request=request, usi

    def formfield_for_manytomany(self, db_field, request=None, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super(MultiDBModelAdmin, self).formfield_for_manytomany(db_field, request=request, usi
```

The implementation provided here implements a multi-database strategy where all objects of a given type are stored on a specific database (e.g., all `User` objects are in the `other` database). If your usage of multiple databases is more complex, your `ModelAdmin` will need to reflect that strategy.

Inlines can be handled in a similar fashion. They require three customized methods:

```
class MultiDBTabularInline(admin.TabularInline):
    using = 'other'

    def queryset(self, request):
        # Tell Django to look for inline objects on the 'other' database.
        return super(MultiDBTabularInline, self).queryset(request).using(self.using)

    def formfield_for_foreignkey(self, db_field, request=None, **kwargs):
        # Tell Django to populate ForeignKey widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_foreignkey(db_field, request=request,

    def formfield_for_manytomany(self, db_field, request=None, **kwargs):
        # Tell Django to populate ManyToMany widgets using a query
        # on the 'other' database.
        return super(MultiDBTabularInline, self).formfield_for_manytomany(db_field, request=request,
```

Once you've written your model admin definitions, they can be registered with any `Admin` instance:

```
from django.contrib import admin

# Specialize the multi-db admin objects for use with specific models.
class BookInline(MultiDBTabularInline):
    model = Book

class PublisherAdmin(MultiDBModelAdmin):
    inlines = [BookInline]

admin.site.register(Author, MultiDBModelAdmin)
admin.site.register(Publisher, PublisherAdmin)

othersite = admin.Site('othersite')
othersite.register(Publisher, MultiDBModelAdmin)
```

This example sets up two admin sites. On the first site, the `Author` and `Publisher` objects are exposed; `Publisher` objects have an tabular inline showing books published by that publisher. The second site exposes just publishers, without the inlines.

Using raw cursors with multiple databases

If you are using more than one database you can use `django.db.connections` to obtain the connection (and cursor) for a specific database. `django.db.connections` is a dictionary-like object that allows you to retrieve a specific connection using its alias:

```
from django.db import connections
cursor = connections['my_db_alias'].cursor()
```

Limitations of multiple databases

Cross-database relations

Django doesn't currently provide any support for foreign key or many-to-many relationships spanning multiple databases. If you have used a router to partition models to different databases, any foreign key and many-to-many relationships defined by those models must be internal to a single database.

This is because of referential integrity. In order to maintain a relationship between two objects, Django needs to know that the primary key of the related object is valid. If the primary key is stored on a separate database, it's not possible to easily evaluate the validity of a primary key.

If you're using Postgres, Oracle, or MySQL with InnoDB, this is enforced at the database integrity level – database level key constraints prevent the creation of relations that can't be validated.

However, if you're using SQLite or MySQL with MyISAM tables, there is no enforced referential integrity; as a result, you may be able to 'fake' cross database foreign keys. However, this configuration is not officially supported by Django.

3.2.8 Tablespaces

A common paradigm for optimizing performance in database systems is the use of tablespaces to organize disk layout.

> **Warning:** Django does not create the tablespaces for you. Please refer to your database engine's documentation for details on creating and managing tablespaces.

Declaring tablespaces for tables

A tablespace can be specified for the table generated by a model by supplying the `db_tablespace` option inside the model's `class Meta`. This option also affects tables automatically created for `ManyToManyFields` in the model.

You can use the `DEFAULT_TABLESPACE` setting to specify a default value for `db_tablespace`. This is useful for setting a tablespace for the built-in Django apps and other applications whose code you cannot control.

Declaring tablespaces for indexes

You can pass the `db_tablespace` option to a `Field` constructor to specify an alternate tablespace for the `Field`'s column index. If no index would be created for the column, the option is ignored.

You can use the `DEFAULT_INDEX_TABLESPACE` setting to specify a default value for `db_tablespace`.

If `db_tablespace` isn't specified and you didn't set `DEFAULT_INDEX_TABLESPACE`, the index is created in the same tablespace as the tables.

An example

```
class TablespaceExample(models.Model):
    name = models.CharField(max_length=30, db_index=True, db_tablespace="indexes")
    data = models.CharField(max_length=255, db_index=True)
    edges = models.ManyToManyField(to="self", db_tablespace="indexes")

    class Meta:
        db_tablespace = "tables"
```

In this example, the tables generated by the `TablespaceExample` model (i.e. the model table and the many-to-many table) would be stored in the `tables` tablespace. The index for the name field and the indexes on the many-to-many table would be stored in the `indexes` tablespace. The `data` field would also generate an index, but no tablespace for it is specified, so it would be stored in the model tablespace `tables` by default.

Database support

PostgreSQL and Oracle support tablespaces. SQLite and MySQL don't.

When you use a backend that lacks support for tablespaces, Django ignores all tablespace-related options. Changed in version 1.4: Since Django 1.4, the PostgreSQL backend supports tablespaces.

3.2.9 Database access optimization

Django's database layer provides various ways to help developers get the most out of their databases. This document gathers together links to the relevant documentation, and adds various tips, organized under a number of headings that outline the steps to take when attempting to optimize your database usage.

Profile first

As general programming practice, this goes without saying. Find out *what queries you are doing and what they are costing you*. You may also want to use an external project like django-debug-toolbar, or a tool that monitors your database directly.

Remember that you may be optimizing for speed or memory or both, depending on your requirements. Sometimes optimizing for one will be detrimental to the other, but sometimes they will help each other. Also, work that is done by the database process might not have the same cost (to you) as the same amount of work done in your Python process. It is up to you to decide what your priorities are, where the balance must lie, and profile all of these as required since this will depend on your application and server.

With everything that follows, remember to profile after every change to ensure that the change is a benefit, and a big enough benefit given the decrease in readability of your code. **All** of the suggestions below come with the caveat that in your circumstances the general principle might not apply, or might even be reversed.

Use standard DB optimization techniques

...including:

- Indexes. This is a number one priority, *after* you have determined from profiling what indexes should be added. Use `django.db.models.Field.db_index` to add these from Django.
- Appropriate use of field types.

We will assume you have done the obvious things above. The rest of this document focuses on how to use Django in such a way that you are not doing unnecessary work. This document also does not address other optimization techniques that apply to all expensive operations, such as *general purpose caching*.

Understand QuerySets

Understanding *QuerySets* is vital to getting good performance with simple code. In particular:

Understand QuerySet evaluation

To avoid performance problems, it is important to understand:

- that *QuerySets are lazy*.
- when *they are evaluated*.
- how *the data is held in memory*.

Understand cached attributes

As well as caching of the whole `QuerySet`, there is caching of the result of attributes on ORM objects. In general, attributes that are not callable will be cached. For example, assuming the *example Weblog models*:

```
>>> entry = Entry.objects.get(id=1)
>>> entry.blog   # Blog object is retrieved at this point
>>> entry.blog   # cached version, no DB access
```

But in general, callable attributes cause DB lookups every time:

```
>>> entry = Entry.objects.get(id=1)
>>> entry.authors.all()   # query performed
>>> entry.authors.all()   # query performed again
```

Be careful when reading template code - the template system does not allow use of parentheses, but will call callables automatically, hiding the above distinction.

Be careful with your own custom properties - it is up to you to implement caching.

Use the `with` template tag

To make use of the caching behavior of `QuerySet`, you may need to use the `with` template tag.

Use `iterator()`

When you have a lot of objects, the caching behavior of the `QuerySet` can cause a large amount of memory to be used. In this case, `iterator()` may help.

Do database work in the database rather than in Python

For instance:

- At the most basic level, use *filter and exclude* to do filtering in the database.

- Use *F() object query expressions* to do filtering against other fields within the same model.
- Use *annotate to do aggregation in the database*.

If these aren't enough to generate the SQL you need:

Use `QuerySet.extra()`

A less portable but more powerful method is `extra()`, which allows some SQL to be explicitly added to the query. If that still isn't powerful enough:

Use raw SQL

Write your own *custom SQL to retrieve data or populate models*. Use `django.db.connection.queries` to find out what Django is writing for you and start from there.

Retrieve everything at once if you know you will need it

Hitting the database multiple times for different parts of a single 'set' of data that you will need all parts of is, in general, less efficient than retrieving it all in one query. This is particularly important if you have a query that is executed in a loop, and could therefore end up doing many database queries, when only one was needed. So:

Use `QuerySet.select_related()` and `prefetch_related()`

Understand `select_related()` and `prefetch_related()` thoroughly, and use them:

- in view code,
- and in *managers and default managers* where appropriate. Be aware when your manager is and is not used; sometimes this is tricky so don't make assumptions.

Don't retrieve things you don't need

Use `QuerySet.values()` and `values_list()`

When you just want a `dict` or `list` of values, and don't need ORM model objects, make appropriate usage of `values()`. These can be useful for replacing model objects in template code - as long as the dicts you supply have the same attributes as those used in the template, you are fine.

Use `QuerySet.defer()` and `only()`

Use `defer()` and `only()` if there are database columns you know that you won't need (or won't need in most cases) to avoid loading them. Note that if you *do* use them, the ORM will have to go and get them in a separate query, making this a pessimization if you use it inappropriately.

Also, be aware that there is some (small extra) overhead incurred inside Django when constructing a model with deferred fields. Don't be too aggressive in deferring fields without profiling as the database has to read most of the non-text, non-VARCHAR data from the disk for a single row in the results, even if it ends up only using a few columns. The `defer()` and `only()` methods are most useful when you can avoid loading a lot of text data or for fields that might take a lot of processing to convert back to Python. As always, profile first, then optimize.

Use QuerySet.count()

...if you only want the count, rather than doing `len(queryset)`.

Use QuerySet.exists()

...if you only want to find out if at least one result exists, rather than `if queryset`.

But:

Don't overuse `count()` and `exists()`

If you are going to need other data from the QuerySet, just evaluate it.

For example, assuming an Email model that has a `body` attribute and a many-to-many relation to User, the following template code is optimal:

```
{% if display_inbox %}
  {% with emails=user.emails.all %}
    {% if emails %}
      <p>You have {{ emails|length }} email(s)</p>
      {% for email in emails %}
        <p>{{ email.body }}</p>
      {% endfor %}
    {% else %}
      <p>No messages today.</p>
    {% endif %}
  {% endwith %}
{% endif %}
```

It is optimal because:

1. Since QuerySets are lazy, this does no database queries if 'display_inbox' is False.
2. Use of `with` means that we store `user.emails.all` in a variable for later use, allowing its cache to be re-used.
3. The line `{% if emails %}` causes `QuerySet.__nonzero__()` to be called, which causes the `user.emails.all()` query to be run on the database, and at the least the first line to be turned into an ORM object. If there aren't any results, it will return False, otherwise True.
4. The use of `{{ emails|length }}` calls `QuerySet.__len__()`, filling out the rest of the cache without doing another query.
5. The `for` loop iterates over the already filled cache.

In total, this code does either one or zero database queries. The only deliberate optimization performed is the use of the `with` tag. Using `QuerySet.exists()` or `QuerySet.count()` at any point would cause additional queries.

Use `QuerySet.update()` and `delete()`

Rather than retrieve a load of objects, set some values, and save them individual, use a bulk SQL UPDATE statement, via *QuerySet.update()*. Similarly, do *bulk deletes* where possible.

Note, however, that these bulk update methods cannot call the `save()` or `delete()` methods of individual instances, which means that any custom behavior you have added for these methods will not be executed, including anything driven from the normal database object *signals*.

Use foreign key values directly

If you only need a foreign key value, use the foreign key value that is already on the object you've got, rather than getting the whole related object and taking its primary key. i.e. do:

```
entry.blog_id
```

instead of:

```
entry.blog.id
```

Insert in bulk

When creating objects, where possible, use the `bulk_create()` method to reduce the number of SQL queries. For example:

```
Entry.objects.bulk_create([
    Entry(headline="Python 3.0 Released"),
    Entry(headline="Python 3.1 Planned")
])
```

...is preferable to:

```
Entry.objects.create(headline="Python 3.0 Released")
Entry.objects.create(headline="Python 3.1 Planned")
```

Note that there are a number of `caveats to this method`, so make sure it's appropriate for your use case.

This also applies to `ManyToManyFields`, so doing:

```
my_band.members.add(me, my_friend)
```

...is preferable to:

```
my_band.members.add(me)
my_band.members.add(my_friend)
```

...where `Bands` and `Artists` have a many-to-many relationship.

3.2.10 Examples of model relationship API usage

Many-to-many relationships

To define a many-to-many relationship, use *ManyToManyField*.

In this example, an `Article` can be published in multiple `Publication` objects, and a `Publication` has multiple `Article` objects:

```
from django.db import models

class Publication(models.Model):
    title = models.CharField(max_length=30)

    def __unicode__(self):
        return self.title

    class Meta:
```

```
        ordering = ('title',)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication)

    def __unicode__(self):
        return self.headline

    class Meta:
        ordering = ('headline',)
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a couple of Publications:

```
>>> p1 = Publication(title='The Python Journal')
>>> p1.save()
>>> p2 = Publication(title='Science News')
>>> p2.save()
>>> p3 = Publication(title='Science Weekly')
>>> p3.save()
```

Create an Article:

```
>>> a1 = Article(headline='Django lets you build Web apps easily')
```

You can't associate it with a Publication until it's been saved:

```
>>> a1.publications.add(p1)
Traceback (most recent call last):
...
ValueError: 'Article' instance needs to have a primary key value before a many-to-many relationship c
```

Save it!

```
>>> a1.save()
```

Associate the Article with a Publication:

```
>>> a1.publications.add(p1)
```

Create another Article, and set it to appear in both Publications:

```
>>> a2 = Article(headline='NASA uses Python')
>>> a2.save()
>>> a2.publications.add(p1, p2)
>>> a2.publications.add(p3)
```

Adding a second time is OK:

```
>>> a2.publications.add(p3)
```

Adding an object of the wrong type raises TypeError:

```
>>> a2.publications.add(a1)
Traceback (most recent call last):
...
TypeError: 'Publication' instance expected
```

Add a Publication directly via publications.add by using keyword arguments:

```
>>> new_publication = a2.publications.create(title='Highlights for Children')
```

Article objects have access to their related Publication objects:

```
>>> a1.publications.all()
[<Publication: The Python Journal>]
>>> a2.publications.all()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>,
```

Publication objects have access to their related Article objects:

```
>>> p2.article_set.all()
[<Article: NASA uses Python>]
>>> p1.article_set.all()
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Publication.objects.get(id=4).article_set.all()
[<Article: NASA uses Python>]
```

Many-to-many relationships can be queried using *lookups across relationships*:

```
>>> Article.objects.filter(publications__id__exact=1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Article.objects.filter(publications__pk=1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Article.objects.filter(publications=1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Article.objects.filter(publications=p1)
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]

>>> Article.objects.filter(publications__title__startswith="Science")
[<Article: NASA uses Python>, <Article: NASA uses Python>]

>>> Article.objects.filter(publications__title__startswith="Science").distinct()
[<Article: NASA uses Python>]
```

The count() function respects distinct() as well:

```
>>> Article.objects.filter(publications__title__startswith="Science").count()
2

>>> Article.objects.filter(publications__title__startswith="Science").distinct().count()
1

>>> Article.objects.filter(publications__in=[1,2]).distinct()
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
>>> Article.objects.filter(publications__in=[p1,p2]).distinct()
[<Article: Django lets you build Web apps easily>, <Article: NASA uses Python>]
```

Reverse m2m queries are supported (i.e., starting at the table that doesn't have a ManyToManyField):

```
>>> Publication.objects.filter(id__exact=1)
[<Publication: The Python Journal>]
>>> Publication.objects.filter(pk=1)
[<Publication: The Python Journal>]

>>> Publication.objects.filter(article__headline__startswith="NASA")
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>,

>>> Publication.objects.filter(article__id__exact=1)
[<Publication: The Python Journal>]
```

```
>>> Publication.objects.filter(article__pk=1)
[<Publication: The Python Journal>]
>>> Publication.objects.filter(article=1)
[<Publication: The Python Journal>]
>>> Publication.objects.filter(article=a1)
[<Publication: The Python Journal>]

>>> Publication.objects.filter(article__in=[1,2]).distinct()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>,
>>> Publication.objects.filter(article__in=[a1,a2]).distinct()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>,
```

Excluding a related item works as you would expect, too (although the SQL involved is a little complex):

```
>>> Article.objects.exclude(publications=p2)
[<Article: Django lets you build Web apps easily>]
```

If we delete a Publication, its Articles won't be able to access it:

```
>>> p1.delete()
>>> Publication.objects.all()
[<Publication: Highlights for Children>, <Publication: Science News>, <Publication: Science Weekly>]
>>> a1 = Article.objects.get(pk=1)
>>> a1.publications.all()
[]
```

If we delete an Article, its Publications won't be able to access it:

```
>>> a2.delete()
>>> Article.objects.all()
[<Article: Django lets you build Web apps easily>]
>>> p2.article_set.all()
[]
```

Adding via the 'other' end of an m2m:

```
>>> a4 = Article(headline='NASA finds intelligent life on Earth')
>>> a4.save()
>>> p2.article_set.add(a4)
>>> p2.article_set.all()
[<Article: NASA finds intelligent life on Earth>]
>>> a4.publications.all()
[<Publication: Science News>]
```

Adding via the other end using keywords:

```
>>> new_article = p2.article_set.create(headline='Oxygen-free diet works wonders')
>>> p2.article_set.all()
[<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>]
>>> a5 = p2.article_set.all()[1]
>>> a5.publications.all()
[<Publication: Science News>]
```

Removing publication from an article:

```
>>> a4.publications.remove(p2)
>>> p2.article_set.all()
[<Article: Oxygen-free diet works wonders>]
>>> a4.publications.all()
[]
```

And from the other end:

```
>>> p2.article_set.remove(a5)
>>> p2.article_set.all()
[]
>>> a5.publications.all()
[]
```

Relation sets can be assigned. Assignment clears any existing set members:

```
>>> a4.publications.all()
[<Publication: Science News>]
>>> a4.publications = [p3]
>>> a4.publications.all()
[<Publication: Science Weekly>]
```

Relation sets can be cleared:

```
>>> p2.article_set.clear()
>>> p2.article_set.all()
[]
```

And you can clear from the other end:

```
>>> p2.article_set.add(a4, a5)
>>> p2.article_set.all()
[<Article: NASA finds intelligent life on Earth>, <Article: Oxygen-free diet works wonders>]
>>> a4.publications.all()
[<Publication: Science News>, <Publication: Science Weekly>]
>>> a4.publications.clear()
>>> a4.publications.all()
[]
>>> p2.article_set.all()
[<Article: Oxygen-free diet works wonders>]
```

Recreate the article and Publication we have deleted:

```
>>> p1 = Publication(title='The Python Journal')
>>> p1.save()
>>> a2 = Article(headline='NASA uses Python')
>>> a2.save()
>>> a2.publications.add(p1, p2, p3)
```

Bulk delete some Publications - references to deleted publications should go:

```
>>> Publication.objects.filter(title__startswith='Science').delete()
>>> Publication.objects.all()
[<Publication: Highlights for Children>, <Publication: The Python Journal>]
>>> Article.objects.all()
[<Article: Django lets you build Web apps easily>, <Article: NASA finds intelligent life on Earth>, <
>>> a2.publications.all()
[<Publication: The Python Journal>]
```

Bulk delete some articles - references to deleted objects should go:

```
>>> q = Article.objects.filter(headline__startswith='Django')
>>> print(q)
[<Article: Django lets you build Web apps easily>]
>>> q.delete()
```

After the delete, the QuerySet cache needs to be cleared, and the referenced objects should be gone:

```
>>> print(q)
[]
>>> p1.article_set.all()
[<Article: NASA uses Python>]
```

An alternate to calling clear() is to assign the empty set:

```
>>> p1.article_set = []
>>> p1.article_set.all()
[]

>>> a2.publications = [p1, new_publication]
>>> a2.publications.all()
[<Publication: Highlights for Children>, <Publication: The Python Journal>]
>>> a2.publications = []
>>> a2.publications.all()
[]
```

Many-to-one relationships

To define a many-to-one relationship, use `ForeignKey`.

```
from django.db import models

class Reporter(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()

    def __unicode__(self):
        return u"%s %s" % (self.first_name, self.last_name)

class Article(models.Model):
    headline = models.CharField(max_length=100)
    pub_date = models.DateField()
    reporter = models.ForeignKey(Reporter)

    def __unicode__(self):
        return self.headline

    class Meta:
        ordering = ('headline',)
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a few Reporters:

```
>>> r = Reporter(first_name='John', last_name='Smith', email='john@example.com')
>>> r.save()

>>> r2 = Reporter(first_name='Paul', last_name='Jones', email='paul@example.com')
>>> r2.save()
```

Create an Article:

```
>>> from datetime import datetime
>>> a = Article(id=None, headline="This is a test", pub_date=datetime(2005, 7, 27), reporter=r)
>>> a.save()
```

```
>>> a.reporter.id
1

>>> a.reporter
<Reporter: John Smith>
```

Article objects have access to their related Reporter objects:

```
>>> r = a.reporter
```

These are strings instead of unicode strings because that's what was used in the creation of this reporter (and we haven't refreshed the data from the database, which always returns unicode strings):

```
>>> r.first_name, r.last_name
('John', 'Smith')
```

Create an Article via the Reporter object:

```
>>> new_article = r.article_set.create(headline="John's second story", pub_date=datetime(2005, 7, 29)
>>> new_article
<Article: John's second story>
>>> new_article.reporter
<Reporter: John Smith>
>>> new_article.reporter.id
1
```

Create a new article, and add it to the article set:

```
>>> new_article2 = Article(headline="Paul's story", pub_date=datetime(2006, 1, 17))
>>> r.article_set.add(new_article2)
>>> new_article2.reporter
<Reporter: John Smith>
>>> new_article2.reporter.id
1
>>> r.article_set.all()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]
```

Add the same article to a different article set - check that it moves:

```
>>> r2.article_set.add(new_article2)
>>> new_article2.reporter.id
2
>>> new_article2.reporter
<Reporter: Paul Jones>
```

Adding an object of the wrong type raises TypeError:

```
>>> r.article_set.add(r2)
Traceback (most recent call last):
...
TypeError: 'Article' instance expected

>>> r.article_set.all()
[<Article: John's second story>, <Article: This is a test>]
>>> r2.article_set.all()
[<Article: Paul's story>]

>>> r.article_set.count()
2
```

```
>>> r2.article_set.count()
1
```

Note that in the last example the article has moved from John to Paul.

Related managers support field lookups as well. The API automatically follows relationships as far as you need. Use double underscores to separate relationships. This works as many levels deep as you want. There's no limit. For example:

```
>>> r.article_set.filter(headline__startswith='This')
[<Article: This is a test>]

# Find all Articles for any Reporter whose first name is "John".
>>> Article.objects.filter(reporter__first_name__exact='John')
[<Article: John's second story>, <Article: This is a test>]
```

Exact match is implied here:

```
>>> Article.objects.filter(reporter__first_name='John')
[<Article: John's second story>, <Article: This is a test>]
```

Query twice over the related field. This translates to an AND condition in the WHERE clause:

```
>>> Article.objects.filter(reporter__first_name__exact='John', reporter__last_name__exact='Smith')
[<Article: John's second story>, <Article: This is a test>]
```

For the related lookup you can supply a primary key value or pass the related object explicitly:

```
>>> Article.objects.filter(reporter__pk=1)
[<Article: John's second story>, <Article: This is a test>]
>>> Article.objects.filter(reporter=1)
[<Article: John's second story>, <Article: This is a test>]
>>> Article.objects.filter(reporter=r)
[<Article: John's second story>, <Article: This is a test>]

>>> Article.objects.filter(reporter__in=[1,2]).distinct()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]
>>> Article.objects.filter(reporter__in=[r,r2]).distinct()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]
```

You can also use a queryset instead of a literal list of instances:

```
>>> Article.objects.filter(reporter__in=Reporter.objects.filter(first_name='John')).distinct()
[<Article: John's second story>, <Article: This is a test>]
```

Querying in the opposite direction:

```
>>> Reporter.objects.filter(article__pk=1)
[<Reporter: John Smith>]
>>> Reporter.objects.filter(article=1)
[<Reporter: John Smith>]
>>> Reporter.objects.filter(article=a)
[<Reporter: John Smith>]

>>> Reporter.objects.filter(article__headline__startswith='This')
[<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]
>>> Reporter.objects.filter(article__headline__startswith='This').distinct()
[<Reporter: John Smith>]
```

Counting in the opposite direction works in conjunction with distinct():

```
>>> Reporter.objects.filter(article__headline__startswith='This').count()
3
>>> Reporter.objects.filter(article__headline__startswith='This').distinct().count()
1
```

Queries can go round in circles:

```
>>> Reporter.objects.filter(article__reporter__first_name__startswith='John')
[<Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>, <Reporter: John Smith>]
>>> Reporter.objects.filter(article__reporter__first_name__startswith='John').distinct()
[<Reporter: John Smith>]
>>> Reporter.objects.filter(article__reporter__exact=r).distinct()
[<Reporter: John Smith>]
```

If you delete a reporter, his articles will be deleted (assuming that the ForeignKey was defined with `django.db.models.ForeignKey.on_delete` set to `CASCADE`, which is the default):

```
>>> Article.objects.all()
[<Article: John's second story>, <Article: Paul's story>, <Article: This is a test>]
>>> Reporter.objects.order_by('first_name')
[<Reporter: John Smith>, <Reporter: Paul Jones>]
>>> r2.delete()
>>> Article.objects.all()
[<Article: John's second story>, <Article: This is a test>]
>>> Reporter.objects.order_by('first_name')
[<Reporter: John Smith>]
```

You can delete using a JOIN in the query:

```
>>> Reporter.objects.filter(article__headline__startswith='This').delete()
>>> Reporter.objects.all()
[]
>>> Article.objects.all()
[]
```

One-to-one relationships

To define a one-to-one relationship, use *OneToOneField*.

In this example, a `Place` optionally can be a `Restaurant`:

```
from django.db import models, transaction, IntegrityError

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

    def __unicode__(self):
        return u"%s the place" % self.name

class Restaurant(models.Model):
    place = models.OneToOneField(Place, primary_key=True)
    serves_hot_dogs = models.BooleanField()
    serves_pizza = models.BooleanField()

    def __unicode__(self):
        return u"%s the restaurant" % self.place.name
```

```
class Waiter(models.Model):
    restaurant = models.ForeignKey(Restaurant)
    name = models.CharField(max_length=50)

    def __unicode__(self):
        return u"%s the waiter at %s" % (self.name, self.restaurant)
```

What follows are examples of operations that can be performed using the Python API facilities.

Create a couple of Places:

```
>>> p1 = Place(name='Demon Dogs', address='944 W. Fullerton')
>>> p1.save()
>>> p2 = Place(name='Ace Hardware', address='1013 N. Ashland')
>>> p2.save()
```

Create a Restaurant. Pass the ID of the "parent" object as this object's ID:

```
>>> r = Restaurant(place=p1, serves_hot_dogs=True, serves_pizza=False)
>>> r.save()
```

A Restaurant can access its place:

```
>>> r.place
<Place: Demon Dogs the place>
```

A Place can access its restaurant, if available:

```
>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>
```

p2 doesn't have an associated restaurant:

```
>>> p2.restaurant
Traceback (most recent call last):
    ...
DoesNotExist: Restaurant matching query does not exist.
```

Set the place using assignment notation. Because place is the primary key on Restaurant, the save will create a new restaurant:

```
>>> r.place = p2
>>> r.save()
>>> p2.restaurant
<Restaurant: Ace Hardware the restaurant>
>>> r.place
<Place: Ace Hardware the place>
```

Set the place back again, using assignment in the reverse direction:

```
>>> p1.restaurant = r
>>> p1.restaurant
<Restaurant: Demon Dogs the restaurant>
```

Restaurant.objects.all() just returns the Restaurants, not the Places. Note that there are two restaurants - Ace Hardware the Restaurant was created in the call to r.place = p2:

```
>>> Restaurant.objects.all()
[<Restaurant: Demon Dogs the restaurant>, <Restaurant: Ace Hardware the restaurant>]
```

Place.objects.all() returns all Places, regardless of whether they have Restaurants:

```
>>> Place.objects.order_by('name')
[<Place: Ace Hardware the place>, <Place: Demon Dogs the place>]
```

You can query the models using *lookups across relationships*:

```
>>> Restaurant.objects.get(place=p1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.get(place__pk=1)
<Restaurant: Demon Dogs the restaurant>
>>> Restaurant.objects.filter(place__name__startswith="Demon")
[<Restaurant: Demon Dogs the restaurant>]
>>> Restaurant.objects.exclude(place__address__contains="Ashland")
[<Restaurant: Demon Dogs the restaurant>]
```

This of course works in reverse:

```
>>> Place.objects.get(pk=1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place__exact=p1)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant=r)
<Place: Demon Dogs the place>
>>> Place.objects.get(restaurant__place__name__startswith="Demon")
<Place: Demon Dogs the place>
```

Add a Waiter to the Restaurant:

```
>>> w = r.waiter_set.create(name='Joe')
>>> w.save()
>>> w
<Waiter: Joe the waiter at Demon Dogs the restaurant>
```

Query the waiters:

```
>>> Waiter.objects.filter(restaurant__place=p1)
[<Waiter: Joe the waiter at Demon Dogs the restaurant>]
>>> Waiter.objects.filter(restaurant__place__name__startswith="Demon")
[<Waiter: Joe the waiter at Demon Dogs the restaurant>]
```

3.3 Handling HTTP requests

Information on handling HTTP requests in Django:

3.3.1 URL dispatcher

A clean, elegant URL scheme is an important detail in a high-quality Web application. Django lets you design URLs however you want, with no framework limitations.

There's no `.php` or `.cgi` required, and certainly none of that `0,2097,1-1-1928,00` nonsense.

See Cool URIs don't change, by World Wide Web creator Tim Berners-Lee, for excellent arguments on why URLs should be clean and usable.

Overview

To design URLs for an app, you create a Python module informally called a **URLconf** (URL configuration). This module is pure Python code and is a simple mapping between URL patterns (as simple regular expressions) to Python callback functions (your views).

This mapping can be as short or as long as needed. It can reference other mappings. And, because it's pure Python code, it can be constructed dynamically. New in version 1.4: Django also allows to translate URLs according to the active language. This process is described in the *internationalization docs*.

How Django processes a request

When a user requests a page from your Django-powered site, this is the algorithm the system follows to determine which Python code to execute:

1. Django determines the root URLconf module to use. Ordinarily, this is the value of the `ROOT_URLCONF` setting, but if the incoming `HttpRequest` object has an attribute called `urlconf` (set by middleware *request processing*), its value will be used in place of the `ROOT_URLCONF` setting.
2. Django loads that Python module and looks for the variable `urlpatterns`. This should be a Python list, in the format returned by the function `django.conf.urls.patterns()`.
3. Django runs through each URL pattern, in order, and stops at the first one that matches the requested URL.
4. Once one of the regexes matches, Django imports and calls the given view, which is a simple Python function. The view gets passed an `HttpRequest` as its first argument and any values captured in the regex as remaining arguments.
5. If no regex matches, or if an exception is raised during any point in this process, Django invokes an appropriate error-handling view. See Error handling below.

Example

Here's a sample URLconf:

```
from django.conf.urls import patterns, url, include

urlpatterns = patterns('',
    (r'^articles/2003/$', 'news.views.special_case_2003'),
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

Notes:

- To capture a value from the URL, just put parenthesis around it.
- There's no need to add a leading slash, because every URL has that. For example, it's `^articles`, not `^/articles`.
- The `'r'` in front of each regular expression string is optional but recommended. It tells Python that a string is "raw" – that nothing in the string should be escaped. See Dive Into Python's explanation.

Example requests:

- A request to `/articles/2005/03/` would match the third entry in the list. Django would call the function `news.views.month_archive(request, '2005', '03')`.

- `/articles/2005/3/` would not match any URL patterns, because the third entry in the list requires two digits for the month.
- `/articles/2003/` would match the first pattern in the list, not the second one, because the patterns are tested in order, and the first one is the first test to pass. Feel free to exploit the ordering to insert special cases like this.
- `/articles/2003` would not match any of these patterns, because each pattern requires that the URL end with a slash.
- `/articles/2003/03/03/` would match the final pattern. Django would call the function `news.views.article_detail(request, '2003', '03', '03')`.

Named groups

The above example used simple, *non-named* regular-expression groups (via parenthesis) to capture bits of the URL and pass them as *positional* arguments to a view. In more advanced usage, it's possible to use *named* regular-expression groups to capture URL bits and pass them as *keyword* arguments to a view.

In Python regular expressions, the syntax for named regular-expression groups is `(?P<name>pattern)`, where `name` is the name of the group and `pattern` is some pattern to match.

Here's the above example URLconf, rewritten to use named groups:

```
urlpatterns = patterns('',
    (r'^articles/2003/$', 'news.views.special_case_2003'),
    (r'^articles/(?P<year>\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', 'news.views.month_archive'),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$', 'news.views.article_detail'),
)
```

This accomplishes exactly the same thing as the previous example, with one subtle difference: The captured values are passed to view functions as keyword arguments rather than positional arguments. For example:

- A request to `/articles/2005/03/` would call the function `news.views.month_archive(request, year='2005', month='03')`, instead of `news.views.month_archive(request, '2005', '03')`.
- A request to `/articles/2003/03/03/` would call the function `news.views.article_detail(request, year='2003', month='03', day='03')`.

In practice, this means your URLconfs are slightly more explicit and less prone to argument-order bugs – and you can reorder the arguments in your views' function definitions. Of course, these benefits come at the cost of brevity; some developers find the named-group syntax ugly and too verbose.

The matching/grouping algorithm

Here's the algorithm the URLconf parser follows, with respect to named groups vs. non-named groups in a regular expression:

If there are any named arguments, it will use those, ignoring non-named arguments. Otherwise, it will pass all non-named arguments as positional arguments.

In both cases, it will pass any extra keyword arguments as keyword arguments. See "Passing extra options to view functions" below.

What the URLconf searches against

The URLconf searches against the requested URL, as a normal Python string. This does not include GET or POST parameters, or the domain name.

For example, in a request to `http://www.example.com/myapp/`, the URLconf will look for `myapp/`.

In a request to `http://www.example.com/myapp/?page=3`, the URLconf will look for `myapp/`.

The URLconf doesn't look at the request method. In other words, all request methods – `POST`, `GET`, `HEAD`, etc. – will be routed to the same function for the same URL.

Syntax of the urlpatterns variable

`urlpatterns` should be a Python list, in the format returned by the function `django.conf.urls.patterns()`. Always use `patterns()` to create the `urlpatterns` variable.

`django.conf.urls` utility functions

Deprecated since version 1.4: Starting with Django 1.4 functions `patterns`, `url`, `include` plus the `handler*` symbols described below live in the `django.conf.urls` module. Until Django 1.3 they were located in `django.conf.urls.defaults`. You still can import them from there but it will be removed in Django 1.6.

patterns

`patterns`(*prefix*, *pattern_description*, ...)

A function that takes a prefix, and an arbitrary number of URL patterns, and returns a list of URL patterns in the format Django needs.

The first argument to `patterns()` is a string `prefix`. See The view prefix below.

The remaining arguments should be tuples in this format:

```
(regular expression, Python callback function [, optional dictionary [, optional name]])
```

...where `optional dictionary` and `optional name` are optional. (See Passing extra options to view functions below.)

Note: Because *patterns()* is a function call, it accepts a maximum of 255 arguments (URL patterns, in this case). This is a limit for all Python function calls. This is rarely a problem in practice, because you'll typically structure your URL patterns modularly by using *include()* sections. However, on the off-chance you do hit the 255-argument limit, realize that *patterns()* returns a Python list, so you can split up the construction of the list.

```
urlpatterns = patterns('',
    ...
    )
urlpatterns += patterns('',
    ...
    )
```

Python lists have unlimited size, so there's no limit to how many URL patterns you can construct. The only limit is that you can only create 254 at a time (the 255th argument is the initial prefix argument).

---

url

url(*regex*, *view*, *kwargs=None*, *name=None*, *prefix=''*)

You can use the `url()` function, instead of a tuple, as an argument to `patterns()`. This is convenient if you want to specify a name without the optional extra arguments dictionary. For example:

```
urlpatterns = patterns('',
    url(r'^index/$', index_view, name="main-view"),
    ...
)
```

This function takes five arguments, most of which are optional:

```
url(regex, view, kwargs=None, name=None, prefix='')
```

See Naming URL patterns for why the `name` parameter is useful.

The `prefix` parameter has the same meaning as the first argument to `patterns()` and is only relevant when you're passing a string as the `view` parameter.

include

include(*<module or pattern_list>*)

A function that takes a full Python import path to another URLconf module that should be "included" in this place.

`include()` also accepts as an argument an iterable that returns URL patterns.

See Including other URLconfs below.

Error handling

When Django can't find a regex matching the requested URL, or when an exception is raised, Django will invoke an error-handling view. The views to use for these cases are specified by three variables which can be set in your root URLconf. Setting these variables in any other URLconf will have no effect.

See the documentation on *customizing error views* for more details.

handler403

handler403

A callable, or a string representing the full Python import path to the view that should be called if the user doesn't have the permissions required to access a resource.

By default, this is `'django.views.defaults.permission_denied'`. That default value should suffice.

See the documentation about *the 403 (HTTP Forbidden) view* for more information. New in version 1.4: `handler403` is new in Django 1.4.

handler404

handler404

A callable, or a string representing the full Python import path to the view that should be called if none of the URL patterns match.

By default, this is `'django.views.defaults.page_not_found'`. That default value should suffice.

handler500

handler500

A callable, or a string representing the full Python import path to the view that should be called in case of server errors. Server errors happen when you have runtime errors in view code.

By default, this is `'django.views.defaults.server_error'`. That default value should suffice.

Notes on capturing text in URLs

Each captured argument is sent to the view as a plain Python string, regardless of what sort of match the regular expression makes. For example, in this URLconf line:

```
(r'^articles/(?P<year>\d{4})/$', 'news.views.year_archive'),
```

...the `year` argument to `news.views.year_archive()` will be a string, not an integer, even though the `\d{4}` will only match integer strings.

A convenient trick is to specify default parameters for your views' arguments. Here's an example URLconf and view:

```
# URLconf
urlpatterns = patterns('',
    (r'^blog/$', 'blog.views.page'),
    (r'^blog/page(?P<num>\d+)/$', 'blog.views.page'),
)

# View (in blog/views.py)
def page(request, num="1"):
    # Output the appropriate page of blog entries, according to num.
```

In the above example, both URL patterns point to the same view – `blog.views.page` – but the first pattern doesn't capture anything from the URL. If the first pattern matches, the `page()` function will use its default argument for `num`, `"1"`. If the second pattern matches, `page()` will use whatever `num` value was captured by the regex.

Performance

Each regular expression in a `urlpatterns` is compiled the first time it's accessed. This makes the system blazingly fast.

The view prefix

You can specify a common prefix in your `patterns()` call, to cut down on code duplication.

Here's the example URLconf from the *Django overview*:

```
from django.conf.urls import patterns, url, include

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
```

```
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

In this example, each view has a common prefix – `'news.views'`. Instead of typing that out for each entry in `urlpatterns`, you can use the first argument to the `patterns()` function to specify a prefix to apply to each view function.

With this in mind, the above example can be written more concisely as:

```
from django.conf.urls import patterns, url, include

urlpatterns = patterns('news.views',
    (r'^articles/(\d{4})/$', 'year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'article_detail'),
)
```

Note that you don't put a trailing dot (`"."`) in the prefix. Django puts that in automatically.

Multiple view prefixes

In practice, you'll probably end up mixing and matching views to the point where the views in your `urlpatterns` won't have a common prefix. However, you can still take advantage of the view prefix shortcut to remove duplication. Just add multiple `patterns()` objects together, like this:

Old:

```
from django.conf.urls import patterns, url, include

urlpatterns = patterns('',
    (r'^$', 'myapp.views.app_index'),
    (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$', 'myapp.views.month_display'),
    (r'^tag/(?P<tag>\w+)/$', 'weblog.views.tag'),
)
```

New:

```
from django.conf.urls import patterns, url, include

urlpatterns = patterns('myapp.views',
    (r'^$', 'app_index'),
    (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$','month_display'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(?P<tag>\w+)/$', 'tag'),
)
```

Including other URLconfs

At any point, your `urlpatterns` can "include" other URLconf modules. This essentially "roots" a set of URLs below other ones.

For example, here's an excerpt of the URLconf for the Django Web site itself. It includes a number of other URLconfs:

```
from django.conf.urls import patterns, url, include
```

```
urlpatterns = patterns('',
    # ... snip ...
    (r'^comments/', include('django.contrib.comments.urls')),
    (r'^community/', include('django_website.aggregator.urls')),
    (r'^contact/', include('django_website.contact.urls')),
    (r'^r/', include('django.conf.urls.shortcut')),
    # ... snip ...
)
```

Note that the regular expressions in this example don't have a `$` (end-of-string match character) but do include a trailing slash. Whenever Django encounters `include()`, it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

Another possibility is to include additional URL patterns not by specifying the URLconf Python module defining them as the include argument but by using directly the pattern list as returned by patterns instead. For example:

```
from django.conf.urls import patterns, url, include

extra_patterns = patterns('',
    url(r'^reports/(?P<id>\d+)/$', 'credit.views.report', name='credit-reports'),
    url(r'^charge/$', 'credit.views.charge', name='credit-charge'),
)

urlpatterns = patterns('',
    url(r'^$', 'apps.main.views.homepage', name='site-homepage'),
    (r'^help/', include('apps.help.urls')),
    (r'^credit/', include(extra_patterns)),
)
```

This approach can be seen in use when you deploy an instance of the Django Admin application. The Django Admin is deployed as instances of a `AdminSite`; each `AdminSite` instance has an attribute `urls` that returns the url patterns available to that instance. It is this attribute that you `include()` into your projects `urlpatterns` when you deploy the admin instance.

Captured parameters

An included URLconf receives any captured parameters from parent URLconfs, so the following example is valid:

```
# In settings/urls/main.py
urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)

# In foo/urls/blog.py
urlpatterns = patterns('foo.views',
    (r'^$', 'blog.index'),
    (r'^archive/$', 'blog.archive'),
)
```

In the above example, the captured `"username"` variable is passed to the included URLconf, as expected.

Defining URL namespaces

When you need to deploy multiple instances of a single application, it can be helpful to be able to differentiate between instances. This is especially important when using *named URL patterns*, since multiple instances of a single application will share named URLs. Namespaces provide a way to tell these named URLs apart.

A URL namespace comes in two parts, both of which are strings:

- An **application namespace**. This describes the name of the application that is being deployed. Every instance of a single application will have the same application namespace. For example, Django's admin application has the somewhat predictable application namespace of `admin`.
- An **instance namespace**. This identifies a specific instance of an application. Instance namespaces should be unique across your entire project. However, an instance namespace can be the same as the application namespace. This is used to specify a default instance of an application. For example, the default Django Admin instance has an instance namespace of `admin`.

URL Namespaces can be specified in two ways.

Firstly, you can provide the application and instance namespace as arguments to `include()` when you construct your URL patterns. For example,:

```
(r'^help/', include('apps.help.urls', namespace='foo', app_name='bar')),
```

This will include the URLs defined in `apps.help.urls` into the application namespace `bar`, with the instance namespace `foo`.

Secondly, you can include an object that contains embedded namespace data. If you `include()` a `patterns` object, that object will be added to the global namespace. However, you can also `include()` an object that contains a 3-tuple containing:

```
(<patterns object>, <application namespace>, <instance namespace>)
```

This will include the nominated URL patterns into the given application and instance namespace. For example, the `urls` attribute of Django's `AdminSite` object returns a 3-tuple that contains all the patterns in an admin site, plus the name of the admin instance, and the application namespace `admin`.

Once you have defined namespaced URLs, you can reverse them. For details on reversing namespaced urls, see the documentation on *reversing namespaced URLs*.

Passing extra options to view functions

URLconfs have a hook that lets you pass extra arguments to your view functions, as a Python dictionary.

Any URLconf tuple can have an optional third element, which should be a dictionary of extra keyword arguments to pass to the view function.

For example:

```
urlpatterns = patterns('blog.views',
    (r'^blog/(?P<year>\d{4})/$', 'year_archive', {'foo': 'bar'}),
)
```

In this example, for a request to `/blog/2005/`, Django will call the `blog.views.year_archive()` view, passing it these keyword arguments:

```
year='2005', foo='bar'
```

This technique is used in the *syndication framework* to pass metadata and options to views.

Dealing with conflicts

It's possible to have a URL pattern which captures named keyword arguments, and also passes arguments with the same names in its dictionary of extra arguments. When this happens, the arguments in the dictionary will be used instead of the arguments captured in the URL.

Passing extra options to `include()`

Similarly, you can pass extra options to `include()`. When you pass extra options to `include()`, *each* line in the included URLconf will be passed the extra options.

For example, these two URLconf sets are functionally identical:

Set one:

```
# main.py
urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)

# inner.py
urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
)
```

Set two:

```
# main.py
urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py
urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
)
```

Note that extra options will *always* be passed to *every* line in the included URLconf, regardless of whether the line's view actually accepts those options as valid. For this reason, this technique is only useful if you're certain that every view in the included URLconf accepts the extra options you're passing.

Passing callable objects instead of strings

Some developers find it more natural to pass the actual Python function object rather than a string containing the path to its module. This alternative is supported – you can pass any callable object as the view.

For example, given this URLconf in "string" notation:

```
urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^contact/$', 'mysite.views.contact'),
)
```

You can accomplish the same thing by passing objects rather than strings. Just be sure to import the objects:

```
from mysite.views import archive, about, contact

urlpatterns = patterns('',
    (r'^archive/$', archive),
    (r'^about/$', about),
    (r'^contact/$', contact),
)
```

The following example is functionally identical. It's just a bit more compact because it imports the module that contains the views, rather than importing each view individually:

```
from mysite import views

urlpatterns = patterns('',
    (r'^archive/$', views.archive),
    (r'^about/$', views.about),
    (r'^contact/$', views.contact),
)
```

The style you use is up to you.

Note that if you use this technique – passing objects rather than strings – the view prefix (as explained in "The view prefix" above) will have no effect.

Naming URL patterns

It's fairly common to use the same view function in multiple URL patterns in your URLconf. For example, these two URL patterns both point to the `archive` view:

```
urlpatterns = patterns('',
    (r'^archive/(\d{4})/$', archive),
    (r'^archive-summary/(\d{4})/$', archive, {'summary': True}),
)
```

This is completely valid, but it leads to problems when you try to do reverse URL matching (through the `permalink()` decorator or the `url` template tag). Continuing this example, if you wanted to retrieve the URL for the `archive` view, Django's reverse URL matcher would get confused, because *two* URL patterns point at that view.

To solve this problem, Django supports **named URL patterns**. That is, you can give a name to a URL pattern in order to distinguish it from other patterns using the same view and parameters. Then, you can use this name in reverse URL matching.

Here's the above example, rewritten to use named URL patterns:

```
urlpatterns = patterns('',
    url(r'^archive/(\d{4})/$', archive, name="full-archive"),
    url(r'^archive-summary/(\d{4})/$', archive, {'summary': True}, "arch-summary"),
)
```

With these names in place (`full-archive` and `arch-summary`), you can target each pattern individually by using its name:

```
{% url 'arch-summary' 1945 %}
{% url 'full-archive' 2007 %}
```

Even though both URL patterns refer to the `archive` view here, using the `name` parameter to `url()` allows you to tell them apart in templates.

The string used for the URL name can contain any characters you like. You are not restricted to valid Python names.

Note: When you name your URL patterns, make sure you use names that are unlikely to clash with any other application's choice of names. If you call your URL pattern `comment`, and another application does the same thing, there's no guarantee which URL will be inserted into your template when you use this name.

Putting a prefix on your URL names, perhaps derived from the application name, will decrease the chances of collision. We recommend something like `myapp-comment` instead of `comment`.

URL namespaces

Namespaced URLs are specified using the `:` operator. For example, the main index page of the admin application is referenced using `admin:index`. This indicates a namespace of `admin`, and a named URL of `index`.

Namespaces can also be nested. The named URL `foo:bar:whiz` would look for a pattern named `whiz` in the namespace `bar` that is itself defined within the top-level namespace `foo`.

When given a namespaced URL (e.g. `myapp:index`) to resolve, Django splits the fully qualified name into parts, and then tries the following lookup:

1. First, Django looks for a matching application namespace (in this example, `myapp`). This will yield a list of instances of that application.
2. If there is a *current* application defined, Django finds and returns the URL resolver for that instance. The *current* application can be specified as an attribute on the template context - applications that expect to have multiple deployments should set the `current_app` attribute on any `Context` or `RequestContext` that is used to render a template.

   The current application can also be specified manually as an argument to the `reverse()` function.
3. If there is no current application. Django looks for a default application instance. The default application instance is the instance that has an instance namespace matching the application namespace (in this example, an instance of the `myapp` called `myapp`).
4. If there is no default application instance, Django will pick the last deployed instance of the application, whatever its instance name may be.
5. If the provided namespace doesn't match an application namespace in step 1, Django will attempt a direct lookup of the namespace as an instance namespace.

If there are nested namespaces, these steps are repeated for each part of the namespace until only the view name is unresolved. The view name will then be resolved into a URL in the namespace that has been found.

To show this resolution strategy in action, consider an example of two instances of `myapp`: one called `foo`, and one called `bar`. `myapp` has a main index page with a URL named *index*. Using this setup, the following lookups are possible:

- If one of the instances is current - say, if we were rendering a utility page in the instance `bar` - `myapp:index` will resolve to the index page of the instance `bar`.
- If there is no current instance - say, if we were rendering a page somewhere else on the site - `myapp:index` will resolve to the last registered instance of `myapp`. Since there is no default instance, the last instance of `myapp` that is registered will be used. This could be `foo` or `bar`, depending on the order they are introduced into the urlpatterns of the project.
- `foo:index` will always resolve to the index page of the instance `foo`.

If there was also a default instance - i.e., an instance named *myapp* - the following would happen:

- If one of the instances is current - say, if we were rendering a utility page in the instance `bar` - `myapp:index` will resolve to the index page of the instance `bar`.
- If there is no current instance - say, if we were rendering a page somewhere else on the site - `myapp:index` will resolve to the index page of the default instance.
- `foo:index` will again resolve to the index page of the instance `foo`.

`django.core.urlresolvers` utility functions

reverse()

If you need to use something similar to the `url` template tag in your code, Django provides the following function (in the `django.core.urlresolvers` module):

reverse(*viewname*[, *urlconf=None*, *args=None*, *kwargs=None*, *current_app=None*])

`viewname` is either the function name (either a function reference, or the string version of the name, if you used that form in `urlpatterns`) or the URL pattern name. Normally, you won't need to worry about the `urlconf` parameter and will only pass in the positional and keyword arguments to use in the URL matching. For example:

```
from django.core.urlresolvers import reverse

def myview(request):
    return HttpResponseRedirect(reverse('arch-summary', args=[1945]))
```

The `reverse()` function can reverse a large variety of regular expression patterns for URLs, but not every possible one. The main restriction at the moment is that the pattern cannot contain alternative choices using the vertical bar (`"|"`) character. You can quite happily use such patterns for matching against incoming URLs and sending them off to views, but you cannot reverse such patterns.

The `current_app` argument allows you to provide a hint to the resolver indicating the application to which the currently executing view belongs. This `current_app` argument is used as a hint to resolve application namespaces into URLs on specific application instances, according to the *namespaced URL resolution strategy*.

You can use `kwargs` instead of `args`. For example:

```
>>> reverse('admin:app_list', kwargs={'app_label': 'auth'})
'/admin/auth/'
```

`args` and `kwargs` cannot be passed to `reverse()` at the same time.

Make sure your views are all correct.

As part of working out which URL names map to which patterns, the `reverse()` function has to import all of your URLconf files and examine the name of each view. This involves importing each view function. If there are *any* errors whilst importing any of your view functions, it will cause `reverse()` to raise an error, even if that view function is not the one you are trying to reverse.

Make sure that any views you reference in your URLconf files exist and can be imported correctly. Do not include lines that reference views you haven't written yet, because those views will not be importable.

Note: The string returned by `reverse()` is already *urlquoted*. For example:

```
>>> reverse('cities', args=[u'Orléans'])
'.../Orl%C3%A9ans/'
```

Applying further encoding (such as `urlquote()` or `urllib.quote`) to the output of `reverse()` may produce undesirable results.

---

reverse_lazy()

New in version 1.4: *Please see the release notes* A lazily evaluated version of reverse().

reverse_lazy(*viewname*[, *urlconf=None*, *args=None*, *kwargs=None*, *current_app=None*])

It is useful for when you need to use a URL reversal before your project's URLConf is loaded. Some common cases where this function is necessary are:

- providing a reversed URL as the `url` attribute of a generic class-based view.
- providing a reversed URL to a decorator (such as the `login_url` argument for the `django.contrib.auth.decorators.permission_required()` decorator).
- providing a reversed URL as a default value for a parameter in a function's signature.

resolve()

The `django.core.urlresolvers.resolve()` function can be used for resolving URL paths to the corresponding view functions. It has the following signature:

resolve(*path*, *urlconf=None*)

`path` is the URL path you want to resolve. As with `reverse()`, you don't need to worry about the `urlconf` parameter. The function returns a `ResolverMatch` object that allows you to access various meta-data about the resolved URL.

If the URL does not resolve, the function raises an `Http404` exception.

class ResolverMatch

func
The view function that would be used to serve the URL

args
The arguments that would be passed to the view function, as parsed from the URL.

kwargs
The keyword arguments that would be passed to the view function, as parsed from the URL.

url_name
The name of the URL pattern that matches the URL.

app_name
The application namespace for the URL pattern that matches the URL.

namespace
The instance namespace for the URL pattern that matches the URL.

namespaces
The list of individual namespace components in the full instance namespace for the URL pattern that matches the URL. i.e., if the namespace is `foo:bar`, then namespaces will be `['foo', 'bar']`.

A `ResolverMatch` object can then be interrogated to provide information about the URL pattern that matches a URL:

```
# Resolve a URL
match = resolve('/some/path/')
# Print the URL pattern that matches the URL
print(match.url_name)
```

A `ResolverMatch` object can also be assigned to a triple:

```
func, args, kwargs = resolve('/some/path/')
```

Changed in version 1.3: Triple-assignment exists for backwards-compatibility. Prior to Django 1.3, `resolve()` returned a triple containing (view function, arguments, keyword arguments); the `ResolverMatch` object (as well as the namespace and pattern information it provides) is not available in earlier Django releases. One possible use of `resolve()` would be to test whether a view would raise a `Http404` error before redirecting to it:

```
from urlparse import urlparse
from django.core.urlresolvers import resolve
from django.http import HttpResponseRedirect, Http404

def myview(request):
    next = request.META.get('HTTP_REFERER', None) or '/'
    response = HttpResponseRedirect(next)

    # modify the request and response as required, e.g. change locale
    # and set corresponding locale cookie

    view, args, kwargs = resolve(urlparse(next)[2])
    kwargs['request'] = request
    try:
        view(*args, **kwargs)
    except Http404:
        return HttpResponseRedirect('/')
    return response
```

permalink()

The `django.db.models.permalink()` decorator is useful for writing short methods that return a full URL path. For example, a model's `get_absolute_url()` method. See `django.db.models.permalink()` for more.

get_script_prefix()

`get_script_prefix``()`

Normally, you should always use `reverse()` or `permalink()` to define URLs within your application. However, if your application constructs part of the URL hierarchy itself, you may occasionally need to generate URLs. In that case, you need to be able to find the base URL of the Django project within its Web server (normally, `reverse()` takes care of this for you). In that case, you can call `get_script_prefix()`, which will return the script prefix portion of the URL for your Django project. If your Django project is at the root of its Web server, this is always `"/"`.

3.3.2 Writing views

A view function, or *view* for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path. There's no other requirement–no "magic," so to speak. For the sake of putting the code *somewhere*, the convention is to put views in a file called `views.py`, placed in your project or application directory.

A simple view

Here's a view that returns the current date and time, as an HTML document:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Let's step through this code one line at a time:

- First, we import the class `HttpResponse` from the `django.http` module, along with Python's `datetime` library.
- Next, we define a function called `current_datetime`. This is the view function. Each view function takes an `HttpRequest` object as its first parameter, which is typically named `request`.

  Note that the name of the view function doesn't matter; it doesn't have to be named in a certain way in order for Django to recognize it. We're calling it `current_datetime` here, because that name clearly indicates what it does.
- The view returns an `HttpResponse` object that contains the generated response. Each view function is responsible for returning an `HttpResponse` object. (There are exceptions, but we'll get to those later.)

Django's Time Zone

Django includes a `TIME_ZONE` setting that defaults to `America/Chicago`. This probably isn't where you live, so you might want to change it in your settings file.

Mapping URLs to views

So, to recap, this view function returns an HTML page that includes the current date and time. To display this view at a particular URL, you'll need to create a *URLconf*; see *URL dispatcher* for instructions.

Returning errors

Returning HTTP error codes in Django is easy. There are subclasses of `HttpResponse` for a number of common HTTP status codes other than 200 (which means *"OK"*). You can find the full list of available subclasses in the *request/response* documentation. Just return an instance of one of those subclasses instead of a normal `HttpResponse` in order to signify an error. For example:

```
def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

There isn't a specialized subclass for every possible HTTP response code, since many of them aren't going to be that common. However, as documented in the `HttpResponse` documentation, you can also pass the HTTP status code into the constructor for `HttpResponse` to create a return class for any status code you like. For example:

```
def my_view(request):
    # ...

    # Return a "created" (201) response code.
    return HttpResponse(status=201)
```

Because 404 errors are by far the most common HTTP error, there's an easier way to handle those errors.

The Http404 exception

class `django.http.`**Http404**

When you return an error such as `HttpResponseNotFound`, you're responsible for defining the HTML of the resulting error page:

```
return HttpResponseNotFound('<h1>Page not found</h1>')
```

For convenience, and because it's a good idea to have a consistent 404 error page across your site, Django provides an `Http404` exception. If you raise `Http404` at any point in a view function, Django will catch it and return the standard error page for your application, along with an HTTP error code 404.

Example usage:

```
from django.http import Http404

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render_to_response('polls/detail.html', {'poll': p})
```

In order to use the `Http404` exception to its fullest, you should create a template that is displayed when a 404 error is raised. This template should be called `404.html` and located in the top level of your template tree.

Customizing error views

The 404 (page not found) view

When you raise an `Http404` exception, Django loads a special view devoted to handling 404 errors. By default, it's the view `django.views.defaults.page_not_found`, which loads and renders the template `404.html`.

This means you need to define a `404.html` template in your root template directory. This template will be used for all 404 errors. The default 404 view will pass one variable to the template: `request_path`, which is the URL that resulted in the error.

The `page_not_found` view should suffice for 99% of Web applications, but if you want to override it, you can specify `handler404` in your URLconf, like so:

```
handler404 = 'mysite.views.my_custom_404_view'
```

Behind the scenes, Django determines the 404 view by looking for `handler404` in your root URLconf, and falling back to `django.views.defaults.page_not_found` if you did not define one.

Four things to note about 404 views:

- The 404 view is also called if Django doesn't find a match after checking every regular expression in the URLconf.
- If you don't define your own 404 view — and simply use the default, which is recommended — you still have one obligation: you must create a `404.html` template in the root of your template directory.
- The 404 view is passed a `RequestContext` and will have access to variables supplied by your `TEMPLATE_CONTEXT_PROCESSORS` setting (e.g., `MEDIA_URL`).

- If DEBUG is set to True (in your settings module), then your 404 view will never be used, and your URLconf will be displayed instead, with some debug information.

The 500 (server error) view

Similarly, Django executes special-case behavior in the case of runtime errors in view code. If a view results in an exception, Django will, by default, call the view django.views.defaults.server_error, which loads and renders the template 500.html.

This means you need to define a 500.html template in your root template directory. This template will be used for all server errors. The default 500 view passes no variables to this template and is rendered with an empty Context to lessen the chance of additional errors.

This server_error view should suffice for 99% of Web applications, but if you want to override the view, you can specify handler500 in your URLconf, like so:

```
handler500 = 'mysite.views.my_custom_error_view'
```

Behind the scenes, Django determines the 500 view by looking for handler500 in your root URLconf, and falling back to django.views.defaults.server_error if you did not define one.

Two things to note about 500 views:

- If you don't define your own 500 view — and simply use the default, which is recommended — you still have one obligation: you must create a 500.html template in the root of your template directory.
- If DEBUG is set to True (in your settings module), then your 500 view will never be used, and the traceback will be displayed instead, with some debug information.

The 403 (HTTP Forbidden) view

New in version 1.4: *Please see the release notes* In the same vein as the 404 and 500 views, Django has a view to handle 403 Forbidden errors. If a view results in a 403 exception then Django will, by default, call the view django.views.defaults.permission_denied.

This view loads and renders the template 403.html in your root template directory, or if this file does not exist, instead serves the text "403 Forbidden", as per **RFC 2616** (the HTTP 1.1 Specification).

It is possible to override django.views.defaults.permission_denied in the same way you can for the 404 and 500 views by specifying a handler403 in your URLconf:

```
handler403 = 'mysite.views.my_custom_permission_denied_view'
```

3.3.3 View decorators

Django provides several decorators that can be applied to views to support various HTTP features.

Allowed HTTP methods

The decorators in django.views.decorators.http can be used to restrict access to views based on the request method. These decorators will return a django.http.HttpResponseNotAllowed if the conditions are not met.

require_http_methods(*request_method_list*)
Decorator to require that a view only accept particular request methods. Usage:

```
from django.views.decorators.http import require_http_methods

@require_http_methods(["GET", "POST"])
def my_view(request):
    # I can assume now that only GET or POST requests make it this far
    # ...
    pass
```

Note that request methods should be in uppercase.

require_GET()
Decorator to require that a view only accept the GET method.

require_POST()
Decorator to require that a view only accept the POST method.

require_safe()
New in version 1.4: *Please see the release notes* Decorator to require that a view only accept the GET and HEAD methods. These methods are commonly considered "safe" because they should not have the significance of taking an action other than retrieving the requested resource.

Note: Django will automatically strip the content of responses to HEAD requests while leaving the headers unchanged, so you may handle HEAD requests exactly like GET requests in your views. Since some software, such as link checkers, rely on HEAD requests, you might prefer using `require_safe` instead of `require_GET`.

Conditional view processing

The following decorators in `django.views.decorators.http` can be used to control caching behavior on particular views.

condition(*etag_func=None*, *last_modified_func=None*)

etag(*etag_func*)

last_modified(*last_modified_func*)
These decorators can be used to generate `ETag` and `Last-Modified` headers; see *conditional view processing*.

GZip compression

The decorators in `django.views.decorators.gzip` control content compression on a per-view basis.

gzip_page()
This decorator compresses content if the browser allows gzip compression. It sets the `Vary` header accordingly, so that caches will base their storage on the `Accept-Encoding` header.

Vary headers

The decorators in `django.views.decorators.vary` can be used to control caching based on specific request headers.

vary_on_cookie(*func*)

vary_on_headers(**headers*)
The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key.

See *using vary headers*.

3.3.4 File Uploads

When Django handles a file upload, the file data ends up placed in `request.FILES` (for more on the `request` object see the documentation for *request and response objects*). This document explains how files are stored on disk and in memory, and how to customize the default behavior.

Basic file uploads

Consider a simple form containing a `FileField`:

```
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file  = forms.FileField()
```

A view handling this form will receive the file data in `request.FILES`, which is a dictionary containing a key for each `FileField` (or `ImageField`, or other `FileField` subclass) in the form. So the data from the above form would be accessible as `request.FILES['file']`.

Note that `request.FILES` will only contain data if the request method was `POST` and the `<form>` that posted the request has the attribute `enctype="multipart/form-data"`. Otherwise, `request.FILES` will be empty.

Most of the time, you'll simply pass the file data from `request` into the form as described in *Binding uploaded files to a form*. This would look something like:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response

# Imaginary function to handle an uploaded file.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render_to_response('upload.html', {'form': form})
```

Notice that we have to pass `request.FILES` into the form's constructor; this is how file data gets bound into a form.

Handling uploaded files

class UploadedFile

The final piece of the puzzle is handling the actual file data from `request.FILES`. Each entry in this dictionary is an `UploadedFile` object – a simple wrapper around an uploaded file. You'll usually use one of these methods to access the uploaded content:

read()

Read the entire uploaded data from the file. Be careful with this method: if the uploaded file is huge it can

overwhelm your system if you try to read it into memory. You'll probably want to use `chunks()` instead; see below.

multiple_chunks()
Returns `True` if the uploaded file is big enough to require reading in multiple chunks. By default this will be any file larger than 2.5 megabytes, but that's configurable; see below.

chunks()
A generator returning chunks of the file. If `multiple_chunks()` is `True`, you should use this method in a loop instead of `read()`.

In practice, it's often easiest simply to use `chunks()` all the time; see the example below.

name
The name of the uploaded file (e.g. `my_file.txt`).

size
The size, in bytes, of the uploaded file.

There are a few other methods and attributes available on `UploadedFile` objects; see UploadedFile objects for a complete reference.

Putting it all together, here's a common way you might handle an uploaded file:

```
def handle_uploaded_file(f):
    with open('some/file/name.txt', 'wb+') as destination:
        for chunk in f.chunks():
            destination.write(chunk)
```

Looping over `UploadedFile.chunks()` instead of using `read()` ensures that large files don't overwhelm your system's memory.

Where uploaded data is stored

Before you save uploaded files, the data needs to be stored somewhere.

By default, if an uploaded file is smaller than 2.5 megabytes, Django will hold the entire contents of the upload in memory. This means that saving the file involves only a read from memory and a write to disk and thus is very fast.

However, if an uploaded file is too large, Django will write the uploaded file to a temporary file stored in your system's temporary directory. On a Unix-like platform this means you can expect Django to generate a file called something like `/tmp/tmpzfp6I6.upload`. If an upload is large enough, you can watch this file grow in size as Django streams the data onto disk.

These specifics – 2.5 megabytes; `/tmp`; etc. – are simply "reasonable defaults". Read on for details on how you can customize or completely replace upload behavior.

Changing upload handler behavior

Three settings control Django's file upload behavior:

FILE_UPLOAD_MAX_MEMORY_SIZE The maximum size, in bytes, for files that will be uploaded into memory. Files larger than `FILE_UPLOAD_MAX_MEMORY_SIZE` will be streamed to disk.

Defaults to 2.5 megabytes.

FILE_UPLOAD_TEMP_DIR The directory where uploaded files larger than `FILE_UPLOAD_MAX_MEMORY_SIZE` will be stored.

Defaults to your system's standard temporary directory (i.e. `/tmp` on most Unix-like systems).

FILE_UPLOAD_PERMISSIONS The numeric mode (i.e. `0644`) to set newly uploaded files to. For more information about what these modes mean, see the documentation for `os.chmod()`.

If this isn't given or is `None`, you'll get operating-system dependent behavior. On most platforms, temporary files will have a mode of `0600`, and files saved from memory will be saved using the system's standard umask.

> **Warning:** If you're not familiar with file modes, please note that the leading `0` is very important: it indicates an octal number, which is the way that modes must be specified. If you try to use `644`, you'll get totally incorrect behavior.
> **Always prefix the mode with a 0.**

FILE_UPLOAD_HANDLERS The actual handlers for uploaded files. Changing this setting allows complete customization – even replacement – of Django's upload process. See upload handlers, below, for details.

Defaults to:

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler",)
```

Which means "try to upload to memory first, then fall back to temporary files."

`UploadedFile` objects

In addition to those inherited from `File`, all `UploadedFile` objects define the following methods/attributes:

`UploadedFile.`**`content_type`**
The content-type header uploaded with the file (e.g. *`text/plain`* or *`application/pdf`*). Like any data supplied by the user, you shouldn't trust that the uploaded file is actually this type. You'll still need to validate that the file contains the content that the content-type header claims – "trust but verify."

`UploadedFile.`**`charset`**
For *`text/*`* content-types, the character set (i.e. `utf8`) supplied by the browser. Again, "trust but verify" is the best policy here.

`UploadedFile.`**`temporary_file_path`**
Only files uploaded onto disk will have this method; it returns the full path to the temporary uploaded file.

Note: Like regular Python files, you can read the file line-by-line simply by iterating over the uploaded file:

```
for line in uploadedfile:
    do_something_with(line)
```

However, *unlike* standard Python files, `UploadedFile` only understands `\n` (also known as "Unix-style") line endings. If you know that you need to handle uploaded files with different line endings, you'll need to do so in your view.

Upload Handlers

When a user uploads a file, Django passes off the file data to an *upload handler* – a small class that handles file data as it gets uploaded. Upload handlers are initially defined in the `FILE_UPLOAD_HANDLERS` setting, which defaults to:

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler",)
```

Together the `MemoryFileUploadHandler` and `TemporaryFileUploadHandler` provide Django's default file upload behavior of reading small files into memory and large ones onto disk.

You can write custom handlers that customize how Django handles files. You could, for example, use custom handlers to enforce user-level quotas, compress data on the fly, render progress bars, and even send data to another storage location directly without storing it locally.

Modifying upload handlers on the fly

Sometimes particular views require different upload behavior. In these cases, you can override upload handlers on a per-request basis by modifying `request.upload_handlers`. By default, this list will contain the upload handlers given by `FILE_UPLOAD_HANDLERS`, but you can modify the list as you would any other list.

For instance, suppose you've written a `ProgressBarUploadHandler` that provides feedback on upload progress to some sort of AJAX widget. You'd add this handler to your upload handlers like this:

```
request.upload_handlers.insert(0, ProgressBarUploadHandler())
```

You'd probably want to use `list.insert()` in this case (instead of `append()`) because a progress bar handler would need to run *before* any other handlers. Remember, the upload handlers are processed in order.

If you want to replace the upload handlers completely, you can just assign a new list:

```
request.upload_handlers = [ProgressBarUploadHandler()]
```

Note: You can only modify upload handlers *before* accessing `request.POST` or `request.FILES` – it doesn't make sense to change upload handlers after upload handling has already started. If you try to modify `request.upload_handlers` after reading from `request.POST` or `request.FILES` Django will throw an error.

Thus, you should always modify uploading handlers as early in your view as possible.

Also, `request.POST` is accessed by `CsrfViewMiddleware` which is enabled by default. This means you will need to use `csrf_exempt()` on your view to allow you to change the upload handlers. You will then need to use `csrf_protect()` on the function that actually processes the request. Note that this means that the handlers may start receiving the file upload before the CSRF checks have been done. Example code:

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def upload_file_view(request):
    request.upload_handlers.insert(0, ProgressBarUploadHandler())
    return _upload_file_view(request)

@csrf_protect
def _upload_file_view(request):
    ... # Process request
```

Writing custom upload handlers

All file upload handlers should be subclasses of `django.core.files.uploadhandler.FileUploadHandler`. You can define upload handlers wherever you wish.

Required methods Custom file upload handlers **must** define the following methods:

FileUploadHandler.receive_data_chunk(self, raw_data, start) Receives a "chunk" of data from the file upload.

`raw_data` is a byte string containing the uploaded data.

`start` is the position in the file where this `raw_data` chunk begins.

The data you return will get fed into the subsequent upload handlers' `receive_data_chunk` methods. In this way, one handler can be a "filter" for other handlers.

Return `None` from `receive_data_chunk` to sort-circuit remaining upload handlers from getting this chunk.. This is useful if you're storing the uploaded data yourself and don't want future handlers to store a copy of the data.

If you raise a `StopUpload` or a `SkipFile` exception, the upload will abort or the file will be completely skipped.

FileUploadHandler.file_complete(self, file_size) Called when a file has finished uploading.

The handler should return an `UploadedFile` object that will be stored in `request.FILES`. Handlers may also return `None` to indicate that the `UploadedFile` object should come from subsequent upload handlers.

Optional methods Custom upload handlers may also define any of the following optional methods or attributes:

FileUploadHandler.chunk_size Size, in bytes, of the "chunks" Django should store into memory and feed into the handler. That is, this attribute controls the size of chunks fed into `FileUploadHandler.receive_data_chunk`.

For maximum performance the chunk sizes should be divisible by `4` and should not exceed 2 GB (2^{31} bytes) in size. When there are multiple chunk sizes provided by multiple handlers, Django will use the smallest chunk size defined by any handler.

The default is $64*2^{10}$ bytes, or 64 KB.

FileUploadHandler.new_file(self, field_name, file_name, content_type, content_length, chars
Callback signaling that a new file upload is starting. This is called before any data has been fed to any upload handlers.

`field_name` is a string name of the file `<input>` field.

`file_name` is the unicode filename that was provided by the browser.

`content_type` is the MIME type provided by the browser – E.g. `'image/jpeg'`.

`content_length` is the length of the image given by the browser. Sometimes this won't be provided and will be `None`.

`charset` is the character set (i.e. `utf8`) given by the browser. Like `content_length`, this sometimes won't be provided.

This method may raise a `StopFutureHandlers` exception to prevent future handlers from handling this file.

FileUploadHandler.upload_complete(self) Callback signaling that the entire upload (all files) has completed.

FileUploadHandler.handle_raw_input(self, input_data, META, content_length, boundary, encod
Allows the handler to completely override the parsing of the raw HTTP input.

`input_data` is a file-like object that supports `read()`-ing.

`META` is the same object as `request.META`.

`content_length` is the length of the data in `input_data`. Don't read more than `content_length` bytes from `input_data`.

`boundary` is the MIME boundary for this request.

`encoding` is the encoding of the request.

Return `None` if you want upload handling to continue, or a tuple of `(POST, FILES)` if you want to return the new data structures suitable for the request directly.

3.3.5 Django shortcut functions

The package `django.shortcuts` collects helper functions and classes that "span" multiple levels of MVC. In other words, these functions/classes introduce controlled coupling for convenience's sake.

render

render(*request, template_name[, dictionary][, context_instance][, content_type][, status][, current_app]*)
New in version 1.3: *Please see the release notes* Combines a given template with a given context dictionary and returns an `HttpResponse` object with that rendered text.

`render()` is the same as a call to `render_to_response()` with a *context_instance* argument that forces the use of a `RequestContext`.

Required arguments

request The request object used to generate this response.

template_name The full name of a template to use or sequence of template names.

Optional arguments

dictionary A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the view will call it just before rendering the template.

context_instance The context instance to render the template with. By default, the template will be rendered with a `RequestContext` instance (filled with values from `request` and `dictionary`).

content_type The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.

status The status code for the response. Defaults to `200`.

current_app A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.

Example

The following example renders the template `myapp/index.html` with the MIME type *`application/xhtml+xml`*:

```
from django.shortcuts import render

def my_view(request):
    # View code here...
    return render(request, 'myapp/index.html', {"foo": "bar"},
        content_type="application/xhtml+xml")
```

This example is equivalent to:

```
from django.http import HttpResponse
from django.template import RequestContext, loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/template.html')
    c = RequestContext(request, {'foo': 'bar'})
    return HttpResponse(t.render(c),
        content_type="application/xhtml+xml")
```

render_to_response

render_to_response(*template_name[, dictionary][, context_instance][, mimetype]*)
Renders a given template with a given context dictionary and returns an `HttpResponse` object with that rendered text.

Required arguments

template_name The full name of a template to use or sequence of template names. If a sequence is given, the first template that exists will be used. See the *template loader documentation* for more information on how templates are found.

Optional arguments

dictionary A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the view will call it just before rendering the template.

context_instance The context instance to render the template with. By default, the template will be rendered with a `Context` instance (filled with values from `dictionary`). If you need to use *context processors*, render the template with a `RequestContext` instance instead. Your code might look something like this:

```
return render_to_response('my_template.html',
                          my_data_dictionary,
                          context_instance=RequestContext(request))
```

mimetype The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.

Example

The following example renders the template `myapp/index.html` with the MIME type *application/xhtml+xml*:

```
from django.shortcuts import render_to_response

def my_view(request):
    # View code here...
    return render_to_response('myapp/index.html', {"foo": "bar"},
        mimetype="application/xhtml+xml")
```

This example is equivalent to:

```
from django.http import HttpResponse
from django.template import Context, loader

def my_view(request):
    # View code here...
    t = loader.get_template('myapp/template.html')
    c = Context({'foo': 'bar'})
    return HttpResponse(t.render(c),
        mimetype="application/xhtml+xml")
```

redirect

redirect (*to*[, *permanent=False*], **args*, ***kwargs*)

Returns an `HttpResponseRedirect` to the appropriate URL for the arguments passed.

The arguments could be:

- A model: the model's *get_absolute_url()* function will be called.
- A view name, possibly with arguments: *urlresolvers.reverse()* will be used to reverse-resolve the name.
- A URL, which will be used as-is for the redirect location.

By default issues a temporary redirect; pass `permanent=True` to issue a permanent redirect

Examples

You can use the `redirect()` function in a number of ways.

1. By passing some object; that object's `get_absolute_url()` method will be called to figure out the redirect URL:

   ```
   from django.shortcuts import redirect

   def my_view(request):
       ...
       object = MyModel.objects.get(...)
       return redirect(object)
   ```

2. By passing the name of a view and optionally some positional or keyword arguments; the URL will be reverse resolved using the `reverse()` method:

   ```
   def my_view(request):
       ...
       return redirect('some-view-name', foo='bar')
   ```

3. By passing a hardcoded URL to redirect to:

```
def my_view(request):
    ...
    return redirect('/some/url/')
```

This also works with full URLs:

```
def my_view(request):
    ...
    return redirect('http://example.com/')
```

By default, `redirect()` returns a temporary redirect. All of the above forms accept a `permanent` argument; if set to `True` a permanent redirect will be returned:

```
def my_view(request):
    ...
    object = MyModel.objects.get(...)
    return redirect(object, permanent=True)
```

get_object_or_404

get_object_or_404(*klass*, **args*, ***kwargs*)

Calls `get()` on a given model manager, but it raises `Http404` instead of the model's `DoesNotExist` exception.

Required arguments

klass A `Model`, `Manager` or `QuerySet` instance from which to get the object.

****kwargs** Lookup parameters, which should be in the format accepted by `get()` and `filter()`.

Example

The following example gets the object with the primary key of 1 from `MyModel`:

```
from django.shortcuts import get_object_or_404

def my_view(request):
    my_object = get_object_or_404(MyModel, pk=1)
```

This example is equivalent to:

```
from django.http import Http404

def my_view(request):
    try:
        my_object = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404
```

Note: As with `get()`, a `MultipleObjectsReturned` exception will be raised if more than one object is found.

get_list_or_404

get_list_or_404(*klass*, **args*, ***kwargs*)

Returns the result of `filter()` on a given model manager, raising `Http404` if the resulting list is empty.

Required arguments

klass A `Model`, `Manager` or `QuerySet` instance from which to get the list.

****kwargs** Lookup parameters, which should be in the format accepted by `get()` and `filter()`.

Example

The following example gets all published objects from `MyModel`:

```
from django.shortcuts import get_list_or_404

def my_view(request):
    my_objects = get_list_or_404(MyModel, published=True)
```

This example is equivalent to:

```
from django.http import Http404

def my_view(request):
    my_objects = list(MyModel.objects.filter(published=True))
    if not my_objects:
        raise Http404
```

3.3.6 Generic views

See *Class-based views*.

3.3.7 Middleware

Middleware is a framework of hooks into Django's request/response processing. It's a light, low-level "plugin" system for globally altering Django's input and/or output.

Each middleware component is responsible for doing some specific function. For example, Django includes a middleware component, `XViewMiddleware`, that adds an `"X-View"` HTTP header to every response to a `HEAD` request.

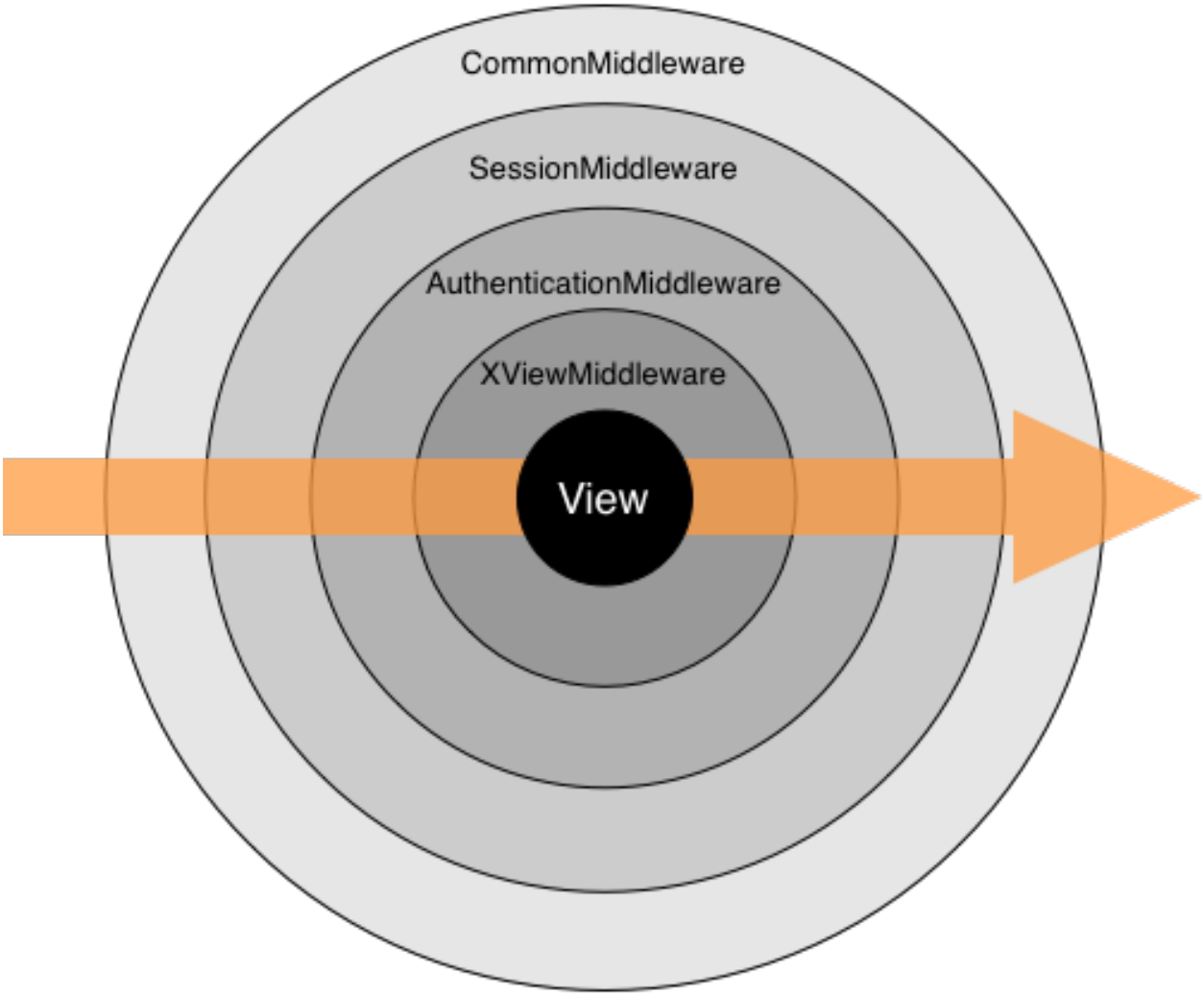
This document explains how middleware works, how you activate middleware, and how to write your own middleware. Django ships with some built-in middleware you can use right out of the box; they're documented in the *built-in middleware reference*.

Activating middleware

To activate a middleware component, add it to the `MIDDLEWARE_CLASSES` list in your Django settings. In `MIDDLEWARE_CLASSES`, each middleware component is represented by a string: the full Python path to the middleware's class name. For example, here's the default `MIDDLEWARE_CLASSES` created by `django-admin.py startproject`:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
)
```

During the request phases (`process_request()` and `process_view()` middleware), Django applies middleware in the order it's defined in `MIDDLEWARE_CLASSES`, top-down. During the response phases (`process_response()` and `process_exception()` middleware), the classes are applied in reverse order, from the bottom up. You can think of it like an onion: each middleware class is a "layer" that wraps the view:



A Django installation doesn't require any middleware – e.g., `MIDDLEWARE_CLASSES` can be empty, if you'd like – but it's strongly suggested that you at least use `CommonMiddleware`.

Writing your own middleware

Writing your own middleware is easy. Each middleware component is a single Python class that defines one or more of the following methods:

process_request

process_request (*self*, *request*)

`request` is an `HttpRequest` object. This method is called on each request, before Django decides which view to execute.

`process_request()` should return either `None` or an `HttpResponse` object. If it returns `None`, Django will continue processing this request, executing any other middleware and, then, the appropriate view. If it returns an `HttpResponse` object, Django won't bother calling ANY other request, view or exception middleware, or the appropriate view; it'll return that `HttpResponse`. Response middleware is always called on every response.

process_view

process_view(*self*, *request*, *view_func*, *view_args*, *view_kwargs*)

`request` is an `HttpRequest` object. `view_func` is the Python function that Django is about to use. (It's the actual function object, not the name of the function as a string.) `view_args` is a list of positional arguments that will be passed to the view, and `view_kwargs` is a dictionary of keyword arguments that will be passed to the view. Neither `view_args` nor `view_kwargs` include the first view argument (`request`).

`process_view()` is called just before Django calls the view. It should return either `None` or an `HttpResponse` object. If it returns `None`, Django will continue processing this request, executing any other `process_view()` middleware and, then, the appropriate view. If it returns an `HttpResponse` object, Django won't bother calling ANY other request, view or exception middleware, or the appropriate view; it'll return that `HttpResponse`. Response middleware is always called on every response.

Note: Accessing `request.POST` or `request.REQUEST` inside middleware from `process_request` or `process_view` will prevent any view running after the middleware from being able to *modify the upload handlers for the request*, and should normally be avoided.

The `CsrfViewMiddleware` class can be considered an exception, as it provides the `csrf_exempt()` and `csrf_protect()` decorators which allow views to explicitly control at what point the CSRF validation should occur.

---

process_template_response

New in version 1.3: *Please see the release notes*

process_template_response(*self*, *request*, *response*)

`request` is an `HttpRequest` object. `response` is a subclass of `SimpleTemplateResponse` (e.g. `TemplateResponse`) or any response object that implements a `render` method.

`process_template_response()` must return a response object that implements a `render` method. It could alter the given `response` by changing `response.template_name` and `response.context_data`, or it could create and return a brand-new `SimpleTemplateResponse` or equivalent.

`process_template_response()` will only be called if the response instance has a `render()` method, indicating that it is a `TemplateResponse` or equivalent.

You don't need to explicitly render responses – responses will be automatically rendered once all template response middleware has been called.

Middleware are run in reverse order during the response phase, which includes process_template_response.

process_response

process_response(*self*, *request*, *response*)

`request` is an `HttpRequest` object. `response` is the `HttpResponse` object returned by a Django view.

`process_response()` must return an `HttpResponse` object. It could alter the given `response`, or it could create and return a brand-new `HttpResponse`.

Unlike the `process_request()` and `process_view()` methods, the `process_response()` method is always called, even if the `process_request()` and `process_view()` methods of the same middleware class were skipped because an earlier middleware method returned an `HttpResponse` (this means that your `process_response()` method cannot rely on setup done in `process_request()`, for example). In addition, during the response phase the classes are applied in reverse order, from the bottom up. This means classes defined at the end of `MIDDLEWARE_CLASSES` will be run first.

process_exception

process_exception(*self*, *request*, *exception*)

`request` is an `HttpRequest` object. `exception` is an `Exception` object raised by the view function.

Django calls `process_exception()` when a view raises an exception. `process_exception()` should return either `None` or an `HttpResponse` object. If it returns an `HttpResponse` object, the response will be returned to the browser. Otherwise, default exception handling kicks in.

Again, middleware are run in reverse order during the response phase, which includes `process_exception`. If an exception middleware returns a response, the middleware classes above that middleware will not be called at all.

__init__

Most middleware classes won't need an initializer since middleware classes are essentially placeholders for the `process_*` methods. If you do need some global state you may use `__init__` to set up. However, keep in mind a couple of caveats:

- Django initializes your middleware without any arguments, so you can't define `__init__` as requiring any arguments.
- Unlike the `process_*` methods which get called once per request, `__init__` gets called only *once*, when the Web server responds to the first request.

Marking middleware as unused It's sometimes useful to determine at run-time whether a piece of middleware should be used. In these cases, your middleware's `__init__` method may raise `django.core.exceptions.MiddlewareNotUsed`. Django will then remove that piece of middleware from the middleware process.

Guidelines

- Middleware classes don't have to subclass anything.
- The middleware class can live anywhere on your Python path. All Django cares about is that the `MIDDLEWARE_CLASSES` setting includes the path to it.
- Feel free to look at *Django's available middleware* for examples.
- If you write a middleware component that you think would be useful to other people, contribute to the community! *Let us know*, and we'll consider adding it to Django.

3.3.8 How to use sessions

Django provides full support for anonymous sessions. The session framework lets you store and retrieve arbitrary data on a per-site-visitor basis. It stores data on the server side and abstracts the sending and receiving of cookies. Cookies contain a session ID – not the data itself (unless you're using the *cookie based backend*).

Enabling sessions

Sessions are implemented via a piece of *middleware*.

To enable session functionality, do the following:

- Edit the `MIDDLEWARE_CLASSES` setting and make sure it contains `'django.contrib.sessions.middleware.SessionMiddleware'`. The default `settings.py` created by `django-admin.py startproject` has `SessionMiddleware` activated.

If you don't want to use sessions, you might as well remove the `SessionMiddleware` line from `MIDDLEWARE_CLASSES` and `'django.contrib.sessions'` from your `INSTALLED_APPS`. It'll save you a small bit of overhead.

Configuring the session engine

By default, Django stores sessions in your database (using the model `django.contrib.sessions.models.Session`). Though this is convenient, in some setups it's faster to store session data elsewhere, so Django can be configured to store session data on your filesystem or in your cache.

Using database-backed sessions

If you want to use a database-backed session, you need to add `'django.contrib.sessions'` to your `INSTALLED_APPS` setting.

Once you have configured your installation, run `manage.py syncdb` to install the single database table that stores session data.

Using cached sessions

For better performance, you may want to use a cache-based session backend.

To store session data using Django's cache system, you'll first need to make sure you've configured your cache; see the *cache documentation* for details.

> **Warning:** You should only use cache-based sessions if you're using the Memcached cache backend. The local-memory cache backend doesn't retain data long enough to be a good choice, and it'll be faster to use file or database sessions directly instead of sending everything through the file or database cache backends.

Once your cache is configured, you've got two choices for how to store data in the cache:

- Set `SESSION_ENGINE` to `"django.contrib.sessions.backends.cache"` for a simple caching session store. Session data will be stored directly your cache. However, session data may not be persistent: cached data can be evicted if the cache fills up or if the cache server is restarted.
- For persistent, cached data, set `SESSION_ENGINE` to `"django.contrib.sessions.backends.cached_db"`. This uses a write-through cache – every write to the cache will also be written to the database. Session reads only use the database if the data is not already in the cache.

Both session stores are quite fast, but the simple cache is faster because it disregards persistence. In most cases, the `cached_db` backend will be fast enough, but if you need that last bit of performance, and are willing to let session data be expunged from time to time, the `cache` backend is for you.

If you use the `cached_db` session backend, you also need to follow the configuration instructions for the using database-backed sessions.

Using file-based sessions

To use file-based sessions, set the `SESSION_ENGINE` setting to `"django.contrib.sessions.backends.file"`.

You might also want to set the `SESSION_FILE_PATH` setting (which defaults to output from `tempfile.gettempdir()`, most likely `/tmp`) to control where Django stores session files. Be sure to check that your Web server has permissions to read and write to this location.

Using cookie-based sessions

New in version 1.4: *Please see the release notes* To use cookies-based sessions, set the `SESSION_ENGINE` setting to `"django.contrib.sessions.backends.signed_cookies"`. The session data will be stored using Django's tools for *cryptographic signing* and the `SECRET_KEY` setting.

Note: It's recommended to leave the `SESSION_COOKIE_HTTPONLY` setting `True` to prevent tampering of the stored data from JavaScript.

Warning: The session data is signed but not encrypted
When using the cookies backend the session data can be read by the client.
A MAC (Message Authentication Code) is used to protect the data against changes by the client, so that the session data will be invalidated when being tampered with. The same invalidation happens if the client storing the cookie (e.g. your user's browser) can't store all of the session cookie and drops data. Even though Django compresses the data, it's still entirely possible to exceed the common limit of 4096 bytes per cookie.
No freshness guarantee
Note also that while the MAC can guarantee the authenticity of the data (that it was generated by your site, and not someone else), and the integrity of the data (that it is all there and correct), it cannot guarantee freshness i.e. that you are being sent back the last thing you sent to the client. This means that for some uses of session data, the cookie backend might open you up to replay attacks. Cookies will only be detected as 'stale' if they are older than your `SESSION_COOKIE_AGE`.
Performance
Finally, the size of a cookie can have an impact on the speed of your site.

Using sessions in views

When `SessionMiddleware` is activated, each `HttpRequest` object – the first argument to any Django view function – will have a `session` attribute, which is a dictionary-like object.

You can read it and write to `request.session` at any point in your view. You can edit it multiple times.

class `backends.base.`**`SessionBase`**
This is the base class for all session objects. It has the following standard dictionary methods:

`__getitem__`(*key*)
Example: `fav_color = request.session['fav_color']`

__setitem__(*key*, *value*)
Example: `request.session['fav_color'] = 'blue'`

__delitem__(*key*)
Example: `del request.session['fav_color']`. This raises `KeyError` if the given `key` isn't already in the session.

__contains__(*key*)
Example: `'fav_color' in request.session`

get(*key*, *default=None*)
Example: `fav_color = request.session.get('fav_color', 'red')`

pop(*key*)
Example: `fav_color = request.session.pop('fav_color')`

keys()

items()

setdefault()

clear()

It also has these methods:

flush()
Delete the current session data from the session and regenerate the session key value that is sent back to the user in the cookie. This is used if you want to ensure that the previous session data can't be accessed again from the user's browser (for example, the `django.contrib.auth.logout()` function calls it).

set_test_cookie()
Sets a test cookie to determine whether the user's browser supports cookies. Due to the way cookies work, you won't be able to test this until the user's next page request. See Setting test cookies below for more information.

test_cookie_worked()
Returns either `True` or `False`, depending on whether the user's browser accepted the test cookie. Due to the way cookies work, you'll have to call `set_test_cookie()` on a previous, separate page request. See Setting test cookies below for more information.

delete_test_cookie()
Deletes the test cookie. Use this to clean up after yourself.

set_expiry(*value*)
Sets the expiration time for the session. You can pass a number of different values:

- If `value` is an integer, the session will expire after that many seconds of inactivity. For example, calling `request.session.set_expiry(300)` would make the session expire in 5 minutes.
- If `value` is a `datetime` or `timedelta` object, the session will expire at that specific date/time.
- If `value` is `0`, the user's session cookie will expire when the user's Web browser is closed.
- If `value` is `None`, the session reverts to using the global session expiry policy.

Reading a session is not considered activity for expiration purposes. Session expiration is computed from the last time the session was *modified*.

get_expiry_age()
Returns the number of seconds until this session expires. For sessions with no custom expiration (or those set to expire at browser close), this will equal `SESSION_COOKIE_AGE`.

get_expiry_date()
Returns the date this session will expire. For sessions with no custom expiration (or those set to expire at browser close), this will equal the date SESSION_COOKIE_AGE seconds from now.

get_expire_at_browser_close()
Returns either True or False, depending on whether the user's session cookie will expire when the user's Web browser is closed.

Session object guidelines

- Use normal Python strings as dictionary keys on request.session. This is more of a convention than a hard-and-fast rule.
- Session dictionary keys that begin with an underscore are reserved for internal use by Django.
- Don't override request.session with a new object, and don't access or set its attributes. Use it like a Python dictionary.

Examples

This simplistic view sets a has_commented variable to True after a user posts a comment. It doesn't let a user post a comment more than once:

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse("You've already commented.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse('Thanks for your comment!')
```

This simplistic view logs in a "member" of the site:

```
def login(request):
    m = Member.objects.get(username=request.POST['username'])
    if m.password == request.POST['password']:
        request.session['member_id'] = m.id
        return HttpResponse("You're logged in.")
    else:
        return HttpResponse("Your username and password didn't match.")
```

...And this one logs a member out, according to login() above:

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse("You're logged out.")
```

The standard django.contrib.auth.logout() function actually does a bit more than this to prevent inadvertent data leakage. It calls the flush() method of request.session. We are using this example as a demonstration of how to work with session objects, not as a full logout() implementation.

Setting test cookies

As a convenience, Django provides an easy way to test whether the user's browser accepts cookies. Just call the `set_test_cookie()` method of `request.session` in a view, and call `test_cookie_worked()` in a subsequent view – not in the same view call.

This awkward split between `set_test_cookie()` and `test_cookie_worked()` is necessary due to the way cookies work. When you set a cookie, you can't actually tell whether a browser accepted it until the browser's next request.

It's good practice to use `delete_test_cookie()` to clean up after yourself. Do this after you've verified that the test cookie worked.

Here's a typical usage example:

```
def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse("You're logged in.")
        else:
            return HttpResponse("Please enable cookies and try again.")
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

Using sessions out of views

An API is available to manipulate session data outside of a view:

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> import datetime
>>> s = SessionStore()
>>> s['last_login'] = datetime.datetime(2005, 8, 20, 13, 35, 10)
>>> s.save()
>>> s.session_key
'2b1189a188b44ad18c35e113ac6ceead'

>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceead')
>>> s['last_login']
datetime.datetime(2005, 8, 20, 13, 35, 0)
```

In order to prevent session fixation attacks, sessions keys that don't exist are regenerated:

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore(session_key='no-such-session-here')
>>> s.save()
>>> s.session_key
'ff882814010ccbc3c870523934fee5a2'
```

If you're using the `django.contrib.sessions.backends.db` backend, each session is just a normal Django model. The `Session` model is defined in `django/contrib/sessions/models.py`. Because it's a normal model, you can access sessions using the normal Django database API:

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

Note that you'll need to call `get_decoded()` to get the session dictionary. This is necessary because the dictionary is stored in an encoded format:

```
>>> s.session_data
'KGRwMQpTJ19hdXRoX3VzZXJfaWQnCnAyCkkxCnMuMTExY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

When sessions are saved

By default, Django only saves to the session database when the session has been modified – that is if any of its dictionary values have been assigned or deleted:

```
# Session is modified.
request.session['foo'] = 'bar'

# Session is modified.
del request.session['foo']

# Session is modified.
request.session['foo'] = {}

# Gotcha: Session is NOT modified, because this alters
# request.session['foo'] instead of request.session.
request.session['foo']['bar'] = 'baz'
```

In the last case of the above example, we can tell the session object explicitly that it has been modified by setting the `modified` attribute on the session object:

```
request.session.modified = True
```

To change this default behavior, set the `SESSION_SAVE_EVERY_REQUEST` setting to `True`. When set to `True`, Django will save the session to the database on every single request.

Note that the session cookie is only sent when a session has been created or modified. If `SESSION_SAVE_EVERY_REQUEST` is `True`, the session cookie will be sent on every request.

Similarly, the `expires` part of a session cookie is updated each time the session cookie is sent. Changed in version 1.5: The session is not saved if the response's status code is 500.

Browser-length sessions vs. persistent sessions

You can control whether the session framework uses browser-length sessions vs. persistent sessions with the `SESSION_EXPIRE_AT_BROWSER_CLOSE` setting.

By default, `SESSION_EXPIRE_AT_BROWSER_CLOSE` is set to `False`, which means session cookies will be stored in users' browsers for as long as `SESSION_COOKIE_AGE`. Use this if you don't want people to have to log in every time they open a browser.

If `SESSION_EXPIRE_AT_BROWSER_CLOSE` is set to `True`, Django will use browser-length cookies – cookies that expire as soon as the user closes his or her browser. Use this if you want people to have to log in every time they open a browser.

This setting is a global default and can be overwritten at a per-session level by explicitly calling the `set_expiry()` method of `request.session` as described above in using sessions in views.

Clearing the session table

If you're using the database backend, note that session data can accumulate in the `django_session` database table and Django does *not* provide automatic purging. Therefore, it's your job to purge expired sessions on a regular basis.

To understand this problem, consider what happens when a user uses a session. When a user logs in, Django adds a row to the `django_session` database table. Django updates this row each time the session data changes. If the user logs out manually, Django deletes the row. But if the user does *not* log out, the row never gets deleted.

Django provides a sample clean-up script: `django-admin.py cleanup`. That script deletes any session in the session table whose `expire_date` is in the past – but your application may have different requirements.

Settings

A few *Django settings* give you control over session behavior:

SESSION_ENGINE

Default: `django.contrib.sessions.backends.db`

Controls where Django stores session data. Valid values are:

- `'django.contrib.sessions.backends.db'`
- `'django.contrib.sessions.backends.file'`
- `'django.contrib.sessions.backends.cache'`
- `'django.contrib.sessions.backends.cached_db'`
- `'django.contrib.sessions.backends.signed_cookies'`

See configuring the session engine for more details.

SESSION_FILE_PATH

Default: `/tmp/`

If you're using file-based session storage, this sets the directory in which Django will store session data.

SESSION_COOKIE_AGE

Default: `1209600` (2 weeks, in seconds)

The age of session cookies, in seconds.

SESSION_COOKIE_DOMAIN

Default: `None`

The domain to use for session cookies. Set this to a string such as `".example.com"` (note the leading dot!) for cross-domain cookies, or use `None` for a standard domain cookie.

SESSION_COOKIE_HTTPONLY

Default: `True`

Whether to use HTTPOnly flag on the session cookie. If this is set to `True`, client-side JavaScript will not to be able to access the session cookie.

HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the **RFC 2109** standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of client side script accessing the protected cookie data.

SESSION_COOKIE_NAME

Default: `'sessionid'`

The name of the cookie to use for sessions. This can be whatever you want.

SESSION_COOKIE_PATH

Default: `'/'`

The path set on the session cookie. This should either match the URL path of your Django installation or be parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths, and each instance will only see its own session cookie.

SESSION_COOKIE_SECURE

Default: `False`

Whether to use a secure cookie for the session cookie. If this is set to `True`, the cookie will be marked as "secure," which means browsers may ensure that the cookie is only sent under an HTTPS connection.

SESSION_EXPIRE_AT_BROWSER_CLOSE

Default: `False`

Whether to expire the session when the user closes his or her browser. See "Browser-length sessions vs. persistent sessions" above.

SESSION_SAVE_EVERY_REQUEST

Default: `False`

Whether to save the session data on every request. If this is `False` (default), then the session data will only be saved if it has been modified – that is, if any of its dictionary values have been assigned or deleted.

Technical details

- The session dictionary should accept any pickleable Python object. See the `pickle` module for more information.
- Session data is stored in a database table named `django_session` .
- Django only sends a cookie if it needs to. If you don't set any session data, it won't send a session cookie.

Session IDs in URLs

The Django sessions framework is entirely, and solely, cookie-based. It does not fall back to putting session IDs in URLs as a last resort, as PHP does. This is an intentional design decision. Not only does that behavior make URLs ugly, it makes your site vulnerable to session-ID theft via the "Referer" header.

3.4 Working with forms

About this document

This document provides an introduction to Django's form handling features. For a more detailed look at specific areas of the forms API, see *The Forms API*, *Form fields*, and *Form and field validation*.

`django.forms` is Django's form-handling library.

While it is possible to process form submissions just using Django's `HttpRequest` class, using the form library takes care of a number of common form-related tasks. Using it, you can:

1. Display an HTML form with automatically generated form widgets.
2. Check submitted data against a set of validation rules.
3. Redisplay a form in the case of validation errors.
4. Convert submitted form data to the relevant Python data types.

3.4.1 Overview

The library deals with these concepts:

Widget A class that corresponds to an HTML form widget, e.g. `<input type="text">` or `<textarea>`. This handles rendering of the widget as HTML.

Field A class that is responsible for doing validation, e.g. an `EmailField` that makes sure its data is a valid email address.

Form A collection of fields that knows how to validate itself and display itself as HTML.

Form Media The CSS and JavaScript resources that are required to render a form.

The library is decoupled from the other Django components, such as the database layer, views and templates. It relies only on Django settings, a couple of `django.utils` helper functions and Django's internationalization hooks (but you're not required to be using internationalization features to use this library).

3.4.2 Form objects

A Form object encapsulates a sequence of form fields and a collection of validation rules that must be fulfilled in order for the form to be accepted. Form classes are created as subclasses of `django.forms.Form` and make use of a declarative style that you'll be familiar with if you've used Django's database models.

For example, consider a form used to implement "contact me" functionality on a personal Web site:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

A form is composed of `Field` objects. In this case, our form has four fields: `subject`, `message`, `sender` and `cc_myself`. `CharField`, `EmailField` and `BooleanField` are just three of the available field types; a full list can be found in *Form fields*.

If your form is going to be used to directly add or edit a Django model, you can use a *ModelForm* to avoid duplicating your model description.

Using a form in a view

The standard pattern for processing a form in a view looks like this:

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

def contact(request):
    if request.method == 'POST': # If the form has been submitted...
        form = ContactForm(request.POST) # A form bound to the POST data
        if form.is_valid(): # All validation rules pass
            # Process the data in form.cleaned_data
            # ...
            return HttpResponseRedirect('/thanks/') # Redirect after POST
    else:
        form = ContactForm() # An unbound form

    return render(request, 'contact.html', {
        'form': form,
    })
```

There are three possible code paths here:

Form submitted?	Data?	What occurs
Unsubmitted	None yet	Template gets passed unbound instance of ContactForm.
Submitted	Invalid data	Template gets passed bound instance of ContactForm.
Submitted	Valid data	Valid data is processed. Redirect to a "thanks" page.

The distinction between *Bound and unbound forms* is important:

- An unbound form has no data associated with it. When rendered to the user, it will be empty or will contain default values.
- A bound form has submitted data, and hence can be used to tell if that data is valid. If an invalid bound form is rendered, it can include inline error messages telling the user what data to correct.

Handling file uploads with a form

To see how to handle file uploads with your form, see *Binding uploaded files to a form*.

Processing the data from a form

Once `is_valid()` returns `True`, the successfully validated form data will be in the `form.cleaned_data` dictionary. This data will have been converted nicely into Python types for you.

Note: You can still access the unvalidated data directly from `request.POST` at this point, but the validated data is better.

In the above example, `cc_myself` will be a boolean value. Likewise, fields such as `IntegerField` and `FloatField` convert values to a Python int and float respectively.

Read-only fields are not available in `form.cleaned_data` (and setting a value in a custom `clean()` method won't have any effect). These fields are displayed as text rather than as input elements, and thus are not posted back to the server.

Extending the earlier example, here's how the form data could be processed:

```
if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    from django.core.mail import send_mail
    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/') # Redirect after POST
```

Tip: For more on sending email from Django, see *Sending email*.

Displaying a form using a template

Forms are designed to work with the Django template language. In the above example, we passed our `ContactForm` instance to the template using the context variable `form`. Here's a simple example template:

```
<form action="/contact/" method="post">{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
```

The form only outputs its own fields; it is up to you to provide the surrounding `<form>` tags and the submit button.

Forms and Cross Site Request Forgery protection

Django ships with an easy-to-use *protection against Cross Site Request Forgeries*. When submitting a form via POST with CSRF protection enabled you must use the `csrf_token` template tag as in the preceding example. However,

since CSRF protection is not directly tied to forms in templates, this tag is omitted from the following examples in this document.

`form.as_p` will output the form with each form field and accompanying label wrapped in a paragraph. Here's the output for our example template:

```
<form action="/contact/" method="post">
<p><label for="id_subject">Subject:</label>
    <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label>
    <input type="text" name="message" id="id_message" /></p>
<p><label for="id_sender">Sender:</label>
    <input type="text" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label>
    <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
<input type="submit" value="Submit" />
</form>
```

Note that each form field has an ID attribute set to `id_<field-name>`, which is referenced by the accompanying label tag. This is important for ensuring forms are accessible to assistive technology such as screen reader software. You can also *customize the way in which labels and ids are generated*.

You can also use `form.as_table` to output table rows (you'll need to provide your own `<table>` tags) and `form.as_ul` to output list items.

Customizing the form template

If the default generated HTML is not to your taste, you can completely customize the way a form is presented using the Django template language. Extending the above example:

```
<form action="/contact/" method="post">
    {{ form.non_field_errors }}
    <div class="fieldWrapper">
        {{ form.subject.errors }}
        <label for="id_subject">Email subject:</label>
        {{ form.subject }}
    </div>
    <div class="fieldWrapper">
        {{ form.message.errors }}
        <label for="id_message">Your message:</label>
        {{ form.message }}
    </div>
    <div class="fieldWrapper">
        {{ form.sender.errors }}
        <label for="id_sender">Your email address:</label>
        {{ form.sender }}
    </div>
    <div class="fieldWrapper">
        {{ form.cc_myself.errors }}
        <label for="id_cc_myself">CC yourself?</label>
        {{ form.cc_myself }}
    </div>
    <p><input type="submit" value="Send message" /></p>
</form>
```

Each named form-field can be output to the template using `{{ form.name_of_field }}`, which will produce the HTML needed to display the form widget. Using `{{ form.name_of_field.errors }}` displays a list of form errors, rendered as an unordered list. This might look like:

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

The list has a CSS class of `errorlist` to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:

```
{% if form.subject.errors %}
    <ol>
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```

Looping over the form's fields

If you're using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a `{% for %}` loop:

```
<form action="/contact/" method="post">
    {% for field in form %}
        <div class="fieldWrapper">
            {{ field.errors }}
            {{ field.label_tag }}: {{ field }}
        </div>
    {% endfor %}
    <p><input type="submit" value="Send message" /></p>
</form>
```

Within this loop, `{{ field }}` is an instance of `BoundField`. `BoundField` also has the following attributes, which can be useful in your templates:

`{{ field.label }}` The label of the field, e.g. `Email address`.

`{{ field.label_tag }}` The field's label wrapped in the appropriate HTML `<label>` tag, e.g. `<label for="id_email">Email address</label>`

`{{ field.value }}` The value of the field. e.g `someone@example.com`

`{{ field.html_name }}` The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set.

`{{ field.help_text }}` Any help text that has been associated with the field.

`{{ field.errors }}` Outputs a `<ul class="errorlist">` containing any validation errors corresponding to this field. You can customize the presentation of the errors with a `{% for error in field.errors %}` loop. In this case, each object in the loop is a simple string containing the error message.

`field.is_hidden` This attribute is `True` if the form field is a hidden field and `False` otherwise. It's not particularly useful as a template variable, but could be useful in conditional tests such as:

```
{% if field.is_hidden %}
   {# Do something special #}
{% endif %}
```

Looping over hidden and visible fields

If you're manually laying out a form in a template, as opposed to relying on Django's default form layout, you might want to treat `<input type="hidden">` fields differently than non-hidden fields. For example, because hidden fields don't display anything, putting error messages "next to" the field could cause confusion for your users – so errors for those fields should be handled differently.

Django provides two methods on a form that allow you to loop over the hidden and visible fields independently: `hidden_fields()` and `visible_fields()`. Here's a modification of an earlier example that uses these two methods:

```
<form action="/contact/" method="post">
    {# Include the hidden fields #}
    {% for hidden in form.hidden_fields %}
    {{ hidden }}
    {% endfor %}
    {# Include the visible fields #}
    {% for field in form.visible_fields %}
        <div class="fieldWrapper">
            {{ field.errors }}
            {{ field.label_tag }}: {{ field }}
        </div>
    {% endfor %}
    <p><input type="submit" value="Send message" /></p>
</form>
```

This example does not handle any errors in the hidden fields. Usually, an error in a hidden field is a sign of form tampering, since normal form interaction won't alter them. However, you could easily insert some error displays for those form errors, as well.

Reusable form templates

If your site uses the same rendering logic for forms in multiple places, you can reduce duplication by saving the form's loop in a standalone template and using the `include` tag to reuse it in other templates:

```
<form action="/contact/" method="post">
    {% include "form_snippet.html" %}
    <p><input type="submit" value="Send message" /></p>
</form>

# In form_snippet.html:

{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }}: {{ field }}
    </div>
{% endfor %}
```

If the form object passed to a template has a different name within the context, you can alias it using the `with` argument of the `include` tag:

```
<form action="/comments/add/" method="post">
    {% include "form_snippet.html" with form=comment_form %}
    <p><input type="submit" value="Submit comment" /></p>
</form>
```

If you find yourself doing this often, you might consider creating a custom *inclusion tag*.

3.4.3 Further topics

This covers the basics, but forms can do a whole lot more:

Formsets

A formset is a layer of abstraction to working with multiple forms on the same page. It can be best compared to a data grid. Let's say you have the following form:

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
```

You might want to allow the user to create several articles at once. To create a formset out of an `ArticleForm` you would do:

```
>>> from django.forms.formsets import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

You now have created a formset named `ArticleFormSet`. The formset gives you the ability to iterate over the forms in the formset and display them as you would with a regular form:

```
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
```

As you can see it only displayed one empty form. The number of empty forms that is displayed is controlled by the `extra` parameter. By default, `formset_factory` defines one extra form; the following example will display two blank forms:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
```

Changed in version 1.3: *Please see the release notes* Prior to Django 1.3, formset instances were not iterable. To render the formset you iterated over the `forms` attribute:

```
>>> formset = ArticleFormSet()
>>> for form in formset.forms:
...     print(form.as_table())
```

Iterating over `formset.forms` will render the forms in the order they were created. The default formset iterator also renders the forms in this order, but you can change this order by providing an alternate implementation for the `__iter__()` method.

Formsets can also be indexed into, which returns the corresponding form. If you override `__iter__`, you will need to also override `__getitem__` to have matching behavior.

Using initial data with a formset

Initial data is what drives the main usability of a formset. As shown above you can define the number of extra forms. What this means is that you are telling the formset how many additional forms to show in addition to the number of forms it generates from the initial data. Lets take a look at an example:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
>>> formset = ArticleFormSet(initial=[
...     {'title': u'Django is now open source',
...      'pub_date': datetime.date.today(),}
... ])

>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" va
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" id
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub
```

There are now a total of three forms showing above. One for the initial data that was passed in and two extra forms. Also note that we are passing in a list of dictionaries as the initial data.

See Also:

Creating formsets from models with model formsets.

Limiting the maximum number of forms

The `max_num` parameter to `formset_factory` gives you the ability to limit the maximum number of empty forms the formset will display:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
>>> formset = ArticleFormset()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
```

If the value of `max_num` is greater than the number of existing objects, up to `extra` additional blank forms will be added to the formset, so long as the total number of forms does not exceed `max_num`.

A `max_num` value of `None` (the default) puts no limit on the number of forms displayed.

Formset validation

Validation with a formset is almost identical to a regular `Form`. There is an `is_valid` method on the formset to provide a convenient way to validate all forms in the formset:

```
>>> ArticleFormSet = formset_factory(ArticleForm)
>>> data = {
...     'form-TOTAL_FORMS': u'1',
...     'form-INITIAL_FORMS': u'0',
...     'form-MAX_NUM_FORMS': u'',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
True
```

We passed in no data to the formset which is resulting in a valid form. The formset is smart enough to ignore extra forms that were not changed. If we provide an invalid article:

```
>>> data = {
...     'form-TOTAL_FORMS': u'2',
...     'form-INITIAL_FORMS': u'0',
...     'form-MAX_NUM_FORMS': u'',
...     'form-0-title': u'Test',
...     'form-0-pub_date': u'1904-06-16',
...     'form-1-title': u'Test',
...     'form-1-pub_date': u'', # <-- this date is missing but required
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {'pub_date': [u'This field is required.']}]
```

As we can see, `formset.errors` is a list whose entries correspond to the forms in the formset. Validation was performed for each of the two forms, and the expected error message appears for the second item. New in version 1.4: *Please see the release notes* We can also check if form data differs from the initial data (i.e. the form was sent without any data):

```
>>> data = {
...     'form-TOTAL_FORMS': u'1',
...     'form-INITIAL_FORMS': u'0',
...     'form-MAX_NUM_FORMS': u'',
...     'form-0-title': u'',
...     'form-0-pub_date': u'',
... }
>>> formset = ArticleFormSet(data)
>>> formset.has_changed()
False
```

Understanding the ManagementForm You may have noticed the additional data (`form-TOTAL_FORMS`, `form-INITIAL_FORMS` and `form-MAX_NUM_FORMS`) that was required in the formset's data above. This data is required for the `ManagementForm`. This form is used by the formset to manage the collection of forms contained in the formset. If you don't provide this management data, an exception will be raised:

```
>>> data = {
...     'form-0-title': u'Test',
...     'form-0-pub_date': u'',
... }
>>> formset = ArticleFormSet(data)
Traceback (most recent call last):
...
django.forms.util.ValidationError: [u'ManagementForm data is missing or has been tampered with']
```

It is used to keep track of how many form instances are being displayed. If you are adding new forms via JavaScript, you should increment the count fields in this form as well.

The management form is available as an attribute of the formset itself. When rendering a formset in a template, you can include all the management data by rendering `{{ my_formset.management_form }}` (substituting the name of your formset as appropriate).

`total_form_count` and `initial_form_count` `BaseFormSet` has a couple of methods that are closely related to the `ManagementForm`, `total_form_count` and `initial_form_count`.

`total_form_count` returns the total number of forms in this formset. `initial_form_count` returns the number of forms in the formset that were pre-filled, and is also used to determine how many forms are required. You

will probably never need to override either of these methods, so please be sure you understand what they do before doing so.

empty_form `BaseFormSet` provides an additional attribute `empty_form` which returns a form instance with a prefix of `__prefix__` for easier use in dynamic forms with JavaScript.

Custom formset validation A formset has a `clean` method similar to the one on a `Form` class. This is where you define your own validation that works at the formset level:

```
>>> from django.forms.formsets import BaseFormSet

>>> class BaseArticleFormSet(BaseFormSet):
...     def clean(self):
...         """Checks that no two articles have the same title."""
...         if any(self.errors):
...             # Don't bother validating the formset unless each form is valid on its own
...             return
...         titles = []
...         for i in range(0, self.total_form_count()):
...             form = self.forms[i]
...             title = form.cleaned_data['title']
...             if title in titles:
...                 raise forms.ValidationError("Articles in a set must have distinct titles.")
...             titles.append(title)

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> data = {
...     'form-TOTAL_FORMS': u'2',
...     'form-INITIAL_FORMS': u'0',
...     'form-MAX_NUM_FORMS': u'',
...     'form-0-title': u'Test',
...     'form-0-pub_date': u'1904-06-16',
...     'form-1-title': u'Test',
...     'form-1-pub_date': u'1912-06-23',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
[u'Articles in a set must have distinct titles.']
```

The formset `clean` method is called after all the `Form.clean` methods have been called. The errors will be found using the `non_form_errors()` method on the formset.

Dealing with ordering and deletion of forms

Common use cases with a formset is dealing with ordering and deletion of the form instances. This has been dealt with for you. The `formset_factory` provides two optional parameters `can_order` and `can_delete` that will do the extra work of adding the extra fields and providing simpler ways of getting to that data.

can_order Default: `False`

Lets you create a formset with the ability to order:

```
>>> ArticleFormSet = formset_factory(ArticleForm, can_order=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" va
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-0-ORDER">Order:</label></th><td><input type="text" name="form-0-ORDER" va
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" va
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub
<tr><th><label for="id_form-1-ORDER">Order:</label></th><td><input type="text" name="form-1-ORDER" va
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub
<tr><th><label for="id_form-2-ORDER">Order:</label></th><td><input type="text" name="form-2-ORDER" id
```

This adds an additional field to each form. This new field is named `ORDER` and is an `forms.IntegerField`. For the forms that came from the initial data it automatically assigned them a numeric value. Let's look at what will happen when the user changes these values:

```
>>> data = {
...     'form-TOTAL_FORMS': u'3',
...     'form-INITIAL_FORMS': u'2',
...     'form-MAX_NUM_FORMS': u'',
...     'form-0-title': u'Article #1',
...     'form-0-pub_date': u'2008-05-10',
...     'form-0-ORDER': u'2',
...     'form-1-title': u'Article #2',
...     'form-1-pub_date': u'2008-05-11',
...     'form-1-ORDER': u'1',
...     'form-2-title': u'Article #3',
...     'form-2-pub_date': u'2008-05-01',
...     'form-2-ORDER': u'0',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> formset.is_valid()
True
>>> for form in formset.ordered_forms:
...     print(form.cleaned_data)
{'pub_date': datetime.date(2008, 5, 1), 'ORDER': 0, 'title': u'Article #3'}
{'pub_date': datetime.date(2008, 5, 11), 'ORDER': 1, 'title': u'Article #2'}
{'pub_date': datetime.date(2008, 5, 10), 'ORDER': 2, 'title': u'Article #1'}
```

`can_delete` Default: `False`

Lets you create a formset with the ability to delete:

```
>>> ArticleFormSet = formset_factory(ArticleForm, can_delete=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset:
```

```
....     print(form.as_table())
<input type="hidden" name="form-TOTAL_FORMS" value="3" id="id_form-TOTAL_FORMS" /><input type="hidden
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" va
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-0-DELETE">Delete:</label></th><td><input type="checkbox" name="form-0-DEL
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" va
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub
<tr><th><label for="id_form-1-DELETE">Delete:</label></th><td><input type="checkbox" name="form-1-DEL
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub
<tr><th><label for="id_form-2-DELETE">Delete:</label></th><td><input type="checkbox" name="form-2-DEL
```

Similar to `can_order` this adds a new field to each form named `DELETE` and is a `forms.BooleanField`. When data comes through marking any of the delete fields you can access them with `deleted_forms`:

```
>>> data = {
...     'form-TOTAL_FORMS': u'3',
...     'form-INITIAL_FORMS': u'2',
...     'form-MAX_NUM_FORMS': u'',
...     'form-0-title': u'Article #1',
...     'form-0-pub_date': u'2008-05-10',
...     'form-0-DELETE': u'on',
...     'form-1-title': u'Article #2',
...     'form-1-pub_date': u'2008-05-11',
...     'form-1-DELETE': u'',
...     'form-2-title': u'',
...     'form-2-pub_date': u'',
...     'form-2-DELETE': u'',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> [form.cleaned_data for form in formset.deleted_forms]
[{'DELETE': True, 'pub_date': datetime.date(2008, 5, 10), 'title': u'Article #1'}]
```

Adding additional fields to a formset

If you need to add additional fields to the formset this can be easily accomplished. The formset base class provides an `add_fields` method. You can simply override this method to add your own fields or even redefine the default fields/attributes of the order and deletion fields:

```
>>> class BaseArticleFormSet(BaseFormSet):
...     def add_fields(self, form, index):
...         super(BaseArticleFormSet, self).add_fields(form, index)
...         form.fields["my_field"] = forms.CharField()

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> formset = ArticleFormSet()
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub
<tr><th><label for="id_form-0-my_field">My field:</label></th><td><input type="text" name="form-0-my_
```

Using a formset in views and templates

Using a formset inside a view is as easy as using a regular `Form` class. The only thing you will want to be aware of is making sure to use the management form inside the template. Let's look at a sample view:

```
def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    if request.method == 'POST':
        formset = ArticleFormSet(request.POST, request.FILES)
        if formset.is_valid():
            # do something with the formset.cleaned_data
            pass
    else:
        formset = ArticleFormSet()
    return render_to_response('manage_articles.html', {'formset': formset})
```

The `manage_articles.html` template might look like this:

```
<form method="post" action="">
    {{ formset.management_form }}
    <table>
        {% for form in formset %}
        {{ form }}
        {% endfor %}
    </table>
</form>
```

However the above can be slightly shortcutted and let the formset itself deal with the management form:

```
<form method="post" action="">
    <table>
        {{ formset }}
    </table>
</form>
```

The above ends up calling the `as_table` method on the formset class.

Manually rendered `can_delete` and `can_order` If you manually render fields in the template, you can render `can_delete` parameter with `{{ form.DELETE }}`:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form.id }}
        <ul>
            <li>{{ form.title }}</li>
            {% if formset.can_delete %}
                <li>{{ form.DELETE }}</li>
            {% endif %}
        </ul>
    {% endfor %}
</form>
```

Similarly, if the formset has the ability to order (`can_order=True`), it is possible to render it with `{{ form.ORDER }}`.

Using more than one formset in a view You are able to use more than one formset in a view if you like. Formsets borrow much of its behavior from forms. With that said you are able to use `prefix` to prefix formset form field

names with a given value to allow more than one formset to be sent to a view without name clashing. Lets take a look at how this might be accomplished:

```
def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    BookFormSet = formset_factory(BookForm)
    if request.method == 'POST':
        article_formset = ArticleFormSet(request.POST, request.FILES, prefix='articles')
        book_formset = BookFormSet(request.POST, request.FILES, prefix='books')
        if article_formset.is_valid() and book_formset.is_valid():
            # do something with the cleaned_data on the formsets.
            pass
    else:
        article_formset = ArticleFormSet(prefix='articles')
        book_formset = BookFormSet(prefix='books')
    return render_to_response('manage_articles.html', {
        'article_formset': article_formset,
        'book_formset': book_formset,
    })
```

You would then render the formsets as normal. It is important to point out that you need to pass `prefix` on both the POST and non-POST cases so that it is rendered and processed correctly.

Creating forms from models

ModelForm

class ModelForm

If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a `BlogComment` model, and you want to create a form that lets people submit comments. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model.

For this reason, Django provides a helper class that let you create a `Form` class from a Django model.

For example:

```
>>> from django.forms import ModelForm

# Create the form class.
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article

# Creating a form to add an article.
>>> form = ArticleForm()

# Creating a form to change an existing article.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Field types The generated `Form` class will have a form field for every model field. Each model field has a corresponding default form field. For example, a `CharField` on a model is represented as a `CharField` on a form. A model `ManyToManyField` is represented as a `MultipleChoiceField`. Here is the full list of conversions:

Model field	Form field
AutoField	Not represented in the form
BigIntegerField	IntegerField with min_value set to -9223372036854775808 and max_value set to 9223372036854775807.
BooleanField	BooleanField
CharField	CharField with max_length set to the model field's max_length
CommaSeparatedIntegerField	CharField
DateField	DateField
DateTimeField	DateTimeField
DecimalField	DecimalField
EmailField	EmailField
FileField	FileField
FilePathField	CharField
FloatField	FloatField
ForeignKey	ModelChoiceField (see below)
ImageField	ImageField
IntegerField	IntegerField
IPAddressField	IPAddressField
GenericIPAddressField	GenericIPAddressField
ManyToManyField	ModelMultipleChoiceField (see below)
NullBooleanField	CharField
PhoneNumberField	USPhoneNumberField (from django.contrib.localflavor.us)
PositiveIntegerField	IntegerField
PositiveSmallIntegerField	IntegerField
SlugField	SlugField
SmallIntegerField	IntegerField
TextField	CharField with widget=forms.Textarea
TimeField	TimeField
URLField	URLField

As you might expect, the ForeignKey and ManyToManyField model field types are special cases:

- ForeignKey is represented by django.forms.ModelChoiceField, which is a ChoiceField whose choices are a model QuerySet.
- ManyToManyField is represented by django.forms.ModelMultipleChoiceField, which is a MultipleChoiceField whose choices are a model QuerySet.

In addition, each generated form field has attributes set as follows:

- If the model field has blank=True, then required is set to False on the form field. Otherwise, required=True.
- The form field's label is set to the verbose_name of the model field, with the first character capitalized.
- The form field's help_text is set to the help_text of the model field.
- If the model field has choices set, then the form field's widget will be set to Select, with choices coming from the model field's choices. The choices will normally include the blank choice which is selected by default. If the field is required, this forces the user to make a selection. The blank choice will not be included if the model field has blank=False and an explicit default value (the default value will be initially selected instead).

Finally, note that you can override the form field used for a given model field. See Overriding the default field types or widgets below.

A full example Consider this set of models:

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = (
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
)

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __unicode__(self):
        return self.name

class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
    class Meta:
        model = Author

class BookForm(ModelForm):
    class Meta:
        model = Book
```

With these models, the `ModelForm` subclasses above would be roughly equivalent to this (the only difference being the `save()` method, which we'll discuss in a moment.):

```
from django import forms

class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(max_length=3,
                widget=forms.Select(choices=TITLE_CHOICES))
    birth_date = forms.DateField(required=False)

class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())
```

The `is_valid()` method and `errors` The first time you call `is_valid()` or access the `errors` attribute of a `ModelForm` triggers *form validation* as well as *model validation*. This has the side-effect of cleaning the model you pass to the `ModelForm` constructor. For instance, calling `is_valid()` on your form will convert any date fields on your model to actual date objects.

The `save()` method Every form produced by `ModelForm` also has a `save()` method. This method creates and saves a database object from the data bound to the form. A subclass of `ModelForm` can accept an existing model instance as the keyword argument `instance`; if this is supplied, `save()` will update that instance. If it's not supplied, `save()` will create a new instance of the specified model:

```
# Create a form instance from POST data.
>>> f = ArticleForm(request.POST)
```

```
# Save a new Article object from the form's data.
>>> new_article = f.save()

# Create a form to edit an existing Article.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(instance=a)
>>> f.save()

# Create a form to edit an existing Article, but use
# POST data to populate the form.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Note that `save()` will raise a `ValueError` if the data in the form doesn't validate – i.e., if form.errors evaluates to True.

This `save()` method accepts an optional `commit` keyword argument, which accepts either `True` or `False`. If you call `save()` with `commit=False`, then it will return an object that hasn't yet been saved to the database. In this case, it's up to you to call `save()` on the resulting model instance. This is useful if you want to do custom processing on the object before saving it, or if you want to use one of the specialized *model saving options*. `commit` is `True` by default.

Another side effect of using `commit=False` is seen when your model has a many-to-many relation with another model. If your model has a many-to-many relation and you specify `commit=False` when you save a form, Django cannot immediately save the form data for the many-to-many relation. This is because it isn't possible to save many-to-many data for an instance until the instance exists in the database.

To work around this problem, every time you save a form using `commit=False`, Django adds a `save_m2m()` method to your `ModelForm` subclass. After you've manually saved the instance produced by the form, you can invoke `save_m2m()` to save the many-to-many form data. For example:

```
# Create a form instance with POST data.
>>> f = AuthorForm(request.POST)

# Create, but don't save the new author instance.
>>> new_author = f.save(commit=False)

# Modify the author in some way.
>>> new_author.some_field = 'some_value'

# Save the new instance.
>>> new_author.save()

# Now, save the many-to-many data for the form.
>>> f.save_m2m()
```

Calling `save_m2m()` is only required if you use `save(commit=False)`. When you use a simple `save()` on a form, all data – including many-to-many data – is saved without the need for any additional method calls. For example:

```
# Create a form instance with POST data.
>>> a = Author()
>>> f = AuthorForm(request.POST, instance=a)

# Create and save the new author instance. There's no need to do anything else.
>>> new_author = f.save()
```

Other than the `save()` and `save_m2m()` methods, a `ModelForm` works exactly the same way as any other `forms` form. For example, the `is_valid()` method is used to check for validity, the `is_multipart()` method is used

to determine whether a form requires multipart file upload (and hence whether `request.FILES` must be passed to the form), etc. See *Binding uploaded files to a form* for more information.

Using a subset of fields on the form In some cases, you may not want all the model fields to appear on the generated form. There are three ways of telling `ModelForm` to use only a subset of the model fields:

1. Set `editable=False` on the model field. As a result, *any* form created from the model via `ModelForm` will not include that field.
2. Use the `fields` attribute of the `ModelForm`'s inner `Meta` class. This attribute, if given, should be a list of field names to include in the form. The order in which the fields names are specified in that list is respected when the form renders them.
3. Use the `exclude` attribute of the `ModelForm`'s inner `Meta` class. This attribute, if given, should be a list of field names to exclude from the form.

For example, if you want a form for the `Author` model (defined above) that includes only the `name` and `title` fields, you would specify `fields` or `exclude` like this:

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title')

class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ('birth_date',)
```

Since the Author model has only 3 fields, 'name', 'title', and 'birth_date', the forms above will contain exactly the same fields.

Note: If you specify `fields` or `exclude` when creating a form with `ModelForm`, then the fields that are not in the resulting form will not be set by the form's `save()` method. Also, if you manually add the excluded fields back to the form, they will not be initialized from the model instance.

Django will prevent any attempt to save an incomplete model, so if the model does not allow the missing fields to be empty, and does not provide a default value for the missing fields, any attempt to `save()` a `ModelForm` with missing fields will fail. To avoid this failure, you must instantiate your model with initial values for the missing, but required fields:

```
author = Author(title='Mr')
form = PartialAuthorForm(request.POST, instance=author)
form.save()
```

Alternatively, you can use `save(commit=False)` and manually set any extra required fields:

```
form = PartialAuthorForm(request.POST)
author = form.save(commit=False)
author.title = 'Mr'
author.save()
```

See the section on saving forms for more details on using `save(commit=False)`.

Overriding the default field types or widgets The default field types, as described in the Field types table above, are sensible defaults. If you have a `DateField` in your model, chances are you'd want that to be represented as a

`DateField` in your form. But `ModelForm` gives you the flexibility of changing the form field type and widget for a given model field.

To specify a custom widget for a field, use the `widgets` attribute of the inner `Meta` class. This should be a dictionary mapping field names to widget classes or instances.

For example, if you want the a `CharField` for the `name` attribute of `Author` to be represented by a `<textarea>` instead of its default `<input type="text">`, you can override the field's widget:

```
from django.forms import ModelForm, Textarea

class AuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title', 'birth_date')
        widgets = {
            'name': Textarea(attrs={'cols': 80, 'rows': 20}),
        }
```

The `widgets` dictionary accepts either widget instances (e.g., `Textarea(...)`) or classes (e.g., `Textarea`).

If you want to further customize a field – including its type, label, etc. – you can do this by declaratively specifying fields like you would in a regular `Form`. Declared fields will override the default ones generated by using the `model` attribute.

For example, if you wanted to use `MyDateFormField` for the `pub_date` field, you could do the following:

```
class ArticleForm(ModelForm):
    pub_date = MyDateFormField()

    class Meta:
        model = Article
```

If you want to override a field's default label, then specify the `label` parameter when declaring the form field:

```
>>> class ArticleForm(ModelForm):
...     pub_date = DateField(label='Publication date')
...
...     class Meta:
...         model = Article
```

Note: If you explicitly instantiate a form field like this, Django assumes that you want to completely define its behavior; therefore, default attributes (such as `max_length` or `required`) are not drawn from the corresponding model. If you want to maintain the behavior specified in the model, you must set the relevant arguments explicitly when declaring the form field.

For example, if the `Article` model looks like this:

```
class Article(models.Model):
    headline = models.CharField(max_length=200, null=True, blank=True,
                                help_text="Use puns liberally")
    content = models.TextField()
```

and you want to do some custom validation for `headline`, while keeping the `blank` and `help_text` values as specified, you might define `ArticleForm` like this:

```
class ArticleForm(ModelForm):
    headline = MyFormField(max_length=200, required=False,
                           help_text="Use puns liberally")
```

```
    class Meta:
        model = Article
```

See the *form field documentation* for more information on fields and their arguments.

Changing the order of fields By default, a `ModelForm` will render fields in the same order that they are defined on the model, with `ManyToManyField` instances appearing last. If you want to change the order in which fields are rendered, you can use the `fields` attribute on the `Meta` class.

The `fields` attribute defines the subset of model fields that will be rendered, and the order in which they will be rendered. For example given this model:

```
class Book(models.Model):
    author = models.ForeignKey(Author)
    title = models.CharField(max_length=100)
```

the `author` field would be rendered first. If we wanted the title field to be rendered first, we could specify the following `ModelForm`:

```
>>> class BookForm(ModelForm):
...     class Meta:
...         model = Book
...         fields = ('title', 'author')
```

Overriding the clean() method You can override the `clean()` method on a model form to provide additional validation in the same way you can on a normal form.

In this regard, model forms have two specific characteristics when compared to forms:

By default the `clean()` method validates the uniqueness of fields that are marked as `unique`, `unique_together` or `unique_for_date|month|year` on the model. Therefore, if you would like to override the `clean()` method and maintain the default validation, you must call the parent class's `clean()` method.

Also, a model form instance bound to a model object will contain a `self.instance` attribute that gives model form methods access to that specific model instance.

Form inheritance As with basic forms, you can extend and reuse `ModelForms` by inheriting them. This is useful if you need to declare extra fields or extra methods on a parent class for use in a number of forms derived from models. For example, using the previous `ArticleForm` class:

```
>>> class EnhancedArticleForm(ArticleForm):
...     def clean_pub_date(self):
...         ...
```

This creates a form that behaves identically to `ArticleForm`, except there's some extra validation and cleaning for the `pub_date` field.

You can also subclass the parent's `Meta` inner class if you want to change the `Meta.fields` or `Meta.excludes` lists:

```
>>> class RestrictedArticleForm(EnhancedArticleForm):
...     class Meta(ArticleForm.Meta):
...         exclude = ('body',)
```

This adds the extra method from the `EnhancedArticleForm` and modifies the original `ArticleForm.Meta` to remove one field.

There are a couple of things to note, however.

- Normal Python name resolution rules apply. If you have multiple base classes that declare a `Meta` inner class, only the first one will be used. This means the child's `Meta`, if it exists, otherwise the `Meta` of the first parent, etc.
- For technical reasons, a subclass cannot inherit from both a `ModelForm` and a `Form` simultaneously.

Chances are these notes won't affect you unless you're trying to do something tricky with subclassing.

Interaction with model validation As part of its validation process, `ModelForm` will call the `clean()` method of each field on your model that has a corresponding field on your form. If you have excluded any model fields, validation will not be run on those fields. See the *form validation* documentation for more on how field cleaning and validation work. Also, your model's `clean()` method will be called before any uniqueness checks are made. See *Validating objects* for more information on the model's `clean()` hook.

Model formsets

Like *regular formsets*, Django provides a couple of enhanced formset classes that make it easy to work with Django models. Let's reuse the `Author` model from above:

```
>>> from django.forms.models import modelformset_factory
>>> AuthorFormSet = modelformset_factory(Author)
```

This will create a formset that is capable of working with the data associated with the `Author` model. It works just like a regular formset:

```
>>> formset = AuthorFormSet()
>>> print(formset)
<input type="hidden" name="form-TOTAL_FORMS" value="1" id="id_form-TOTAL_FORMS" /><input type="hidden
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name" type="text" name
<tr><th><label for="id_form-0-title">Title:</label></th><td><select name="form-0-title" id="id_form-0
<option value="" selected="selected">---------</option>
<option value="MR">Mr.</option>
<option value="MRS">Mrs.</option>
<option value="MS">Ms.</option>
</select></td></tr>
<tr><th><label for="id_form-0-birth_date">Birth date:</label></th><td><input type="text" name="form-0
```

Note: `modelformset_factory` uses `formset_factory` to generate formsets. This means that a model formset is just an extension of a basic formset that knows how to interact with a particular model.

Changing the queryset By default, when you create a formset from a model, the formset will use a queryset that includes all objects in the model (e.g., `Author.objects.all()`). You can override this behavior by using the `queryset` argument:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
```

Alternatively, you can create a subclass that sets `self.queryset` in `__init__`:

```
from django.forms.models import BaseModelFormSet

class BaseAuthorFormSet(BaseModelFormSet):
    def __init__(self, *args, **kwargs):
```

```
        super(BaseAuthorFormSet, self).__init__(*args, **kwargs)
        self.queryset = Author.objects.filter(name__startswith='O')
```

Then, pass your `BaseAuthorFormSet` class to the factory function:

```
>>> AuthorFormSet = modelformset_factory(Author, formset=BaseAuthorFormSet)
```

If you want to return a formset that doesn't include *any* pre-existing instances of the model, you can specify an empty QuerySet:

```
>>> AuthorFormSet(queryset=Author.objects.none())
```

Controlling which fields are used with `fields` and `exclude` By default, a model formset uses all fields in the model that are not marked with `editable=False`. However, this can be overridden at the formset level:

```
>>> AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
```

Using `fields` restricts the formset to use only the given fields. Alternatively, you can take an "opt-out" approach, specifying which fields to exclude:

```
>>> AuthorFormSet = modelformset_factory(Author, exclude=('birth_date',))
```

Providing initial values New in version 1.4: *Please see the release notes* As with regular formsets, it's possible to *specify initial data* for forms in the formset by specifying an `initial` parameter when instantiating the model formset class returned by `modelformset_factory`. However, with model formsets, the initial values only apply to extra forms, those that aren't bound to an existing object instance.

Saving objects in the formset As with a `ModelForm`, you can save the data as a model object. This is done with the formset's `save()` method:

```
# Create a formset instance with POST data.
>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.
>>> instances = formset.save()
```

The `save()` method returns the instances that have been saved to the database. If a given instance's data didn't change in the bound data, the instance won't be saved to the database and won't be included in the return value (`instances`, in the above example).

When fields are missing from the form (for example because they have been excluded), these fields will not be set by the `save()` method. You can find more information about this restriction, which also holds for regular `ModelForms`, in Using a subset of fields on the form.

Pass `commit=False` to return the unsaved model instances:

```
# don't save to the database
>>> instances = formset.save(commit=False)
>>> for instance in instances:
...     # do something with instance
...     instance.save()
```

This gives you the ability to attach data to the instances before saving them to the database. If your formset contains a `ManyToManyField`, you'll also need to call `formset.save_m2m()` to ensure the many-to-many relationships are saved properly.

Limiting the number of editable objects As with regular formsets, you can use the `max_num` and `extra` parameters to `modelformset_factory` to limit the number of extra forms displayed.

`max_num` does not prevent existing objects from being displayed:

```
>>> Author.objects.order_by('name')
[<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]

>>> AuthorFormSet = modelformset_factory(Author, max_num=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> [x.name for x in formset.get_queryset()]
[u'Charles Baudelaire', u'Paul Verlaine', u'Walt Whitman']
```

If the value of `max_num` is greater than the number of existing related objects, up to `extra` additional blank forms will be added to the formset, so long as the total number of forms does not exceed `max_num`:

```
>>> AuthorFormSet = modelformset_factory(Author, max_num=4, extra=2)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> for form in formset:
...     print(form.as_table())
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name" type="text" name
<tr><th><label for="id_form-1-name">Name:</label></th><td><input id="id_form-1-name" type="text" name
<tr><th><label for="id_form-2-name">Name:</label></th><td><input id="id_form-2-name" type="text" name
<tr><th><label for="id_form-3-name">Name:</label></th><td><input id="id_form-3-name" type="text" name
```

A `max_num` value of `None` (the default) puts no limit on the number of forms displayed.

Using a model formset in a view Model formsets are very similar to formsets. Let's say we want to present a formset to edit `Author` model instances:

```
def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author)
    if request.method == 'POST':
        formset = AuthorFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # do something.
    else:
        formset = AuthorFormSet()
    return render_to_response("manage_authors.html", {
        "formset": formset,
    })
```

As you can see, the view logic of a model formset isn't drastically different than that of a "normal" formset. The only difference is that we call `formset.save()` to save the data into the database. (This was described above, in *Saving objects in the formset*.)

Overiding `clean()` on a `model_formset` Just like with `ModelForms`, by default the `clean()` method of a `model_formset` will validate that none of the items in the formset violate the unique constraints on your model (either `unique`, `unique_together` or `unique_for_date|month|year`). If you want to override the `clean()` method on a `model_formset` and maintain this validation, you must call the parent class's `clean` method:

```
class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()
        # example custom validation across forms in the formset:
```

```
        for form in self.forms:
            # your custom formset validation
```

Using a custom queryset As stated earlier, you can override the default queryset used by the model formset:

```
def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author)
    if request.method == "POST":
        formset = AuthorFormSet(request.POST, request.FILES,
                                queryset=Author.objects.filter(name__startswith='O'))
        if formset.is_valid():
            formset.save()
            # Do something.
    else:
        formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
    return render_to_response("manage_authors.html", {
        "formset": formset,
    })
```

Note that we pass the `queryset` argument in both the `POST` and `GET` cases in this example.

Using the formset in the template There are three ways to render a formset in a Django template.

First, you can let the formset do most of the work:

```
<form method="post" action="">
    {{ formset }}
</form>
```

Second, you can manually render the formset, but let the form deal with itself:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
</form>
```

When you manually render the forms yourself, be sure to render the management form as shown above. See the *management form documentation*.

Third, you can manually render each field:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
        {% for field in form %}
            {{ field.label_tag }}: {{ field }}
        {% endfor %}
    {% endfor %}
</form>
```

If you opt to use this third method and you don't iterate over the fields with a `{% for %}` loop, you'll need to render the primary key field. For example, if you were rendering the `name` and `age` fields of a model:

```
<form method="post" action="">
    {{ formset.management_form }}
    {% for form in formset %}
```

```
        {{ form.id }}
        <ul>
            <li>{{ form.name }}</li>
            <li>{{ form.age }}</li>
        </ul>
    {% endfor %}
</form>
```

Notice how we need to explicitly render `{{ form.id }}`. This ensures that the model formset, in the `POST` case, will work correctly. (This example assumes a primary key named `id`. If you've explicitly defined your own primary key that isn't called `id`, make sure it gets rendered.)

Inline formsets

Inline formsets is a small abstraction layer on top of model formsets. These simplify the case of working with related objects via a foreign key. Suppose you have these two models:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author)
    title = models.CharField(max_length=100)
```

If you want to create a formset that allows you to edit books belonging to a particular author, you could do this:

```
>>> from django.forms.models import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book)
>>> author = Author.objects.get(name=u'Mike Royko')
>>> formset = BookFormSet(instance=author)
```

Note: `inlineformset_factory` uses `modelformset_factory` and marks `can_delete=True`.

See Also:

Manually rendered can_delete and can_order.

More than one foreign key to the same model If your model contains more than one foreign key to the same model, you'll need to resolve the ambiguity manually using `fk_name`. For example, consider the following model:

```
class Friendship(models.Model):
    from_friend = models.ForeignKey(Friend)
    to_friend = models.ForeignKey(Friend)
    length_in_months = models.IntegerField()
```

To resolve this, you can use `fk_name` to `inlineformset_factory`:

```
>>> FriendshipFormSet = inlineformset_factory(Friend, Friendship, fk_name="from_friend")
```

Using an inline formset in a view You may want to provide a view that allows a user to edit the related objects of a model. Here's how you can do that:

```
def manage_books(request, author_id):
    author = Author.objects.get(pk=author_id)
    BookInlineFormSet = inlineformset_factory(Author, Book)
```

```
    if request.method == "POST":
        formset = BookInlineFormSet(request.POST, request.FILES, instance=author)
        if formset.is_valid():
            formset.save()
            # Do something. Should generally end with a redirect. For example:
            return HttpResponseRedirect(author.get_absolute_url())
    else:
        formset = BookInlineFormSet(instance=author)
    return render_to_response("manage_books.html", {
        "formset": formset,
    })
```

Notice how we pass `instance` in both the `POST` and `GET` cases.

Form Media

Rendering an attractive and easy-to-use Web form requires more than just HTML - it also requires CSS stylesheets, and if you want to use fancy "Web2.0" widgets, you may also need to include some JavaScript on each page. The exact combination of CSS and JavaScript that is required for any given page will depend upon the widgets that are in use on that page.

This is where Django media definitions come in. Django allows you to associate different media files with the forms and widgets that require that media. For example, if you want to use a calendar to render DateFields, you can define a custom Calendar widget. This widget can then be associated with the CSS and JavaScript that is required to render the calendar. When the Calendar widget is used on a form, Django is able to identify the CSS and JavaScript files that are required, and provide the list of file names in a form suitable for easy inclusion on your Web page.

Media and Django Admin

The Django Admin application defines a number of customized widgets for calendars, filtered selections, and so on. These widgets define media requirements, and the Django Admin uses the custom widgets in place of the Django defaults. The Admin templates will only include those media files that are required to render the widgets on any given page.

If you like the widgets that the Django Admin application uses, feel free to use them in your own application! They're all stored in `django.contrib.admin.widgets`.

Which JavaScript toolkit?

Many JavaScript toolkits exist, and many of them include widgets (such as calendar widgets) that can be used to enhance your application. Django has deliberately avoided blessing any one JavaScript toolkit. Each toolkit has its own relative strengths and weaknesses - use whichever toolkit suits your requirements. Django is able to integrate with any JavaScript toolkit.

Media as a static definition

The easiest way to define media is as a static definition. Using this method, the media declaration is an inner class. The properties of the inner class define the media requirements.

Here's a simple example:

```
class CalendarWidget(forms.TextInput):
    class Media:
        css = {
```

```
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

This code defines a `CalendarWidget`, which will be based on `TextInput`. Every time the CalendarWidget is used on a form, that form will be directed to include the CSS file `pretty.css`, and the JavaScript files `animations.js` and `actions.js`.

This static media definition is converted at runtime into a widget property named `media`. The media for a CalendarWidget instance can be retrieved through this property:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
```

Here's a list of all possible `Media` options. There are no required options.

`css` A dictionary describing the CSS files required for various forms of output media.

The values in the dictionary should be a tuple/list of file names. See the section on media paths for details of how to specify paths to media files. The keys in the dictionary are the output media types. These are the same types accepted by CSS files in media declarations: 'all', 'aural', 'braille', 'embossed', 'handheld', 'print', 'projection', 'screen', 'tty' and 'tv'. If you need to have different stylesheets for different media types, provide a list of CSS files for each output medium. The following example would provide two CSS options – one for the screen, and one for print:

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'print': ('newspaper.css',)
    }
```

If a group of CSS files are appropriate for multiple output media types, the dictionary key can be a comma separated list of output media types. In the following example, TV's and projectors will have the same media requirements:

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'tv,projector': ('lo_res.css',),
        'print': ('newspaper.css',)
    }
```

If this last CSS definition were to be rendered, it would become the following HTML:

```
<link href="http://media.example.com/pretty.css" type="text/css" media="screen" rel="stylesheet" />
<link href="http://media.example.com/lo_res.css" type="text/css" media="tv,projector" rel="stylesheet
<link href="http://media.example.com/newspaper.css" type="text/css" media="print" rel="stylesheet" />
```

`js` A tuple describing the required JavaScript files. See the section on media paths for details of how to specify paths to media files.

`extend` A boolean defining inheritance behavior for media declarations.

By default, any object using a static media definition will inherit all the media associated with the parent widget. This occurs regardless of how the parent defines its media requirements. For example, if we were to extend our basic Calendar widget from the example above:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<link href="http://media.example.com/fancy.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

The FancyCalendar widget inherits all the media from it's parent widget. If you don't want media to be inherited in this way, add an `extend=False` declaration to the media declaration:

```
>>> class FancyCalendarWidget(CalendarWidget):
...     class Media:
...         extend = False
...         css = {
...             'all': ('fancy.css',)
...         }
...         js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print(w.media)
<link href="http://media.example.com/fancy.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

If you require even more control over media inheritance, define your media using a dynamic property. Dynamic properties give you complete control over which media files are inherited, and which are not.

Media as a dynamic property

If you need to perform some more sophisticated manipulation of media requirements, you can define the media property directly. This is done by defining a widget property that returns an instance of `forms.Media`. The constructor for `forms.Media` accepts `css` and `js` keyword arguments in the same format as that used in a static media definition.

For example, the static media definition for our Calendar Widget could also be defined in a dynamic fashion:

```
class CalendarWidget(forms.TextInput):
    def _media(self):
        return forms.Media(css={'all': ('pretty.css',)},
                           js=('animations.js', 'actions.js'))
    media = property(_media)
```

See the section on Media objects for more details on how to construct return values for dynamic media properties.

Paths in media definitions

Changed in version 1.3: *Please see the release notes* Paths used to specify media can be either relative or absolute. If a path starts with '/', 'http://' or 'https://', it will be interpreted as an absolute path, and left as-is. All other paths will be prepended with the value of the appropriate prefix.

As part of the introduction of the *staticfiles app* two new settings were added to refer to "static files" (images, CSS, Javascript, etc.) that are needed to render a complete web page: `STATIC_URL` and `STATIC_ROOT`.

To find the appropriate prefix to use, Django will check if the `STATIC_URL` setting is not `None` and automatically fall back to using `MEDIA_URL`. For example, if the `MEDIA_URL` for your site was `'http://uploads.example.com/'` and `STATIC_URL` was `None`:

```
>>> class CalendarWidget(forms.TextInput):
...     class Media:
...         css = {
...             'all': ('/css/pretty.css',),
...         }
...         js = ('animations.js', 'http://othersite.com/actions.js')

>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://uploads.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

But if `STATIC_URL` is `'http://static.example.com/'`:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://static.example.com/animations.js"></script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

Media objects

When you interrogate the media attribute of a widget or form, the value that is returned is a `forms.Media` object. As we have already seen, the string representation of a Media object is the HTML required to include media in the `<head>` block of your HTML page.

However, Media objects have some other interesting properties.

Media subsets If you only want media of a particular type, you can use the subscript operator to filter out a medium of interest. For example:

```
>>> w = CalendarWidget()
>>> print(w.media)
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>

>>> print(w.media)['css']
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
```

When you use the subscript operator, the value that is returned is a new Media object – but one that only contains the media of interest.

Combining media objects Media objects can also be added together. When two media objects are added, the resulting Media object contains the union of the media from both files:

```
>>> class CalendarWidget(forms.TextInput):
...     class Media:
```

```
...         css = {
...             'all': ('pretty.css',)
...         }
...         js = ('animations.js', 'actions.js')

>>> class OtherWidget(forms.TextInput):
...     class Media:
...         js = ('whizbang.js',)

>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print(w1.media + w2.media)
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

Media on Forms

Widgets aren't the only objects that can have media definitions – forms can also define media. The rules for media definitions on forms are the same as the rules for widgets: declarations can be static or dynamic; path and inheritance rules for those declarations are exactly the same.

Regardless of whether you define a media declaration, *all* Form objects have a media property. The default value for this property is the result of adding the media definitions for all widgets that are part of the form:

```
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)

>>> f = ContactForm()
>>> f.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

If you want to associate additional media with a form – for example, CSS for form layout – simply add a media declaration to the form:

```
>>> class ContactForm(forms.Form):
...     date = DateField(widget=CalendarWidget)
...     name = CharField(max_length=40, widget=OtherWidget)
...
...     class Media:
...         css = {
...             'all': ('layout.css',)
...         }

>>> f = ContactForm()
>>> f.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel="stylesheet" />
<link href="http://media.example.com/layout.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

See Also:

The Forms Reference Covers the full API reference, including form fields, form widgets, and form and field validation.

3.5 The Django template language

About this document

This document explains the language syntax of the Django template system. If you're looking for a more technical perspective on how it works and how to extend it, see *The Django template language: For Python programmers*.

Django's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML. If you have any exposure to other text-based template languages, such as Smarty or CheetahTemplate, you should feel right at home with Django's templates.

Philosophy

If you have a background in programming, or if you're used to languages like PHP which mix programming code directly into HTML, you'll want to bear in mind that the Django template system is not simply Python embedded into HTML. This is by design: the template system is meant to express presentation, not program logic.

The Django template system provides tags which function similarly to some programming constructs – an `if` tag for boolean tests, a `for` tag for looping, etc. – but these are not simply executed as the corresponding Python code, and the template system will not execute arbitrary Python expressions. Only the tags, filters and syntax listed below are supported by default (although you can add *your own extensions* to the template language as needed).

3.5.1 Templates

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains **variables**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. Each element will be explained later in this document.:

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Philosophy

Why use a text-based template instead of an XML-based one (like Zope's TAL)? We wanted Django's template language to be usable for more than just XML/HTML templates. At World Online, we use it for emails, JavaScript and CSV. You can use the template language for any text-based format.

Oh, and one more thing: Making humans edit XML is sadistic!

3.5.2 Variables

Variables look like this: `{{ variable }}`. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore (`"_"`). The dot (`"."`) also appears in variable sections, although that has a special meaning, as indicated below. Importantly, *you cannot have spaces or punctuation characters in variable names.*

Use a dot (`.`) to access attributes of a variable.

Behind the scenes

Technically, when the template system encounters a dot, it tries the following lookups, in this order:

- Dictionary lookup
- Attribute lookup
- Method call
- List-index lookup

This can cause some unexpected behavior with objects that override dictionary lookup. For example, consider the following code snippet that attempts to loop over a `collections.defaultdict`:

```
{% for k, v in defaultdict.iteritems %}
    Do something with k and v here...
{% endfor %}
```

Because dictionary lookup happens first, that behavior kicks in and provides a default value instead of using the intended `.iteritems()` method. In this case, consider converting to a dictionary first.

In the above example, `{{ section.title }}` will be replaced with the `title` attribute of the `section` object.

If you use a variable that doesn't exist, the template system will insert the value of the `TEMPLATE_STRING_IF_INVALID` setting, which is set to `''` (the empty string) by default.

3.5.3 Filters

You can modify variables for display by using **filters**.

Filters look like this: `{{ name|lower }}`. This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase. Use a pipe (`|`) to apply a filter.

Filters can be "chained." The output of one filter is applied to the next. `{{ text|escape|linebreaks }}` is a common idiom for escaping text contents, then converting line breaks to `<p>` tags.

Some filters take arguments. A filter argument looks like this: `{{ bio|truncatewords:30 }}`. This will display the first 30 words of the `bio` variable.

Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaced you'd use `{{ list|join:", " }}`.

Django provides about thirty built-in template filters. You can read all about them in the *built-in filter reference*. To give you a taste of what's available, here are some of the more commonly used template filters:

`default` If a variable is false or empty, use given default. Otherwise, use the value of the variable

For example:

```
{{ value|default:"nothing" }}
```

If `value` isn't provided or is empty, the above will display "`nothing`".

`length` Returns the length of the value. This works for both strings and lists; for example:

```
{{ value|length }}
```

If `value` is `['a', 'b', 'c', 'd']`, the output will be `4`.

`striptags` Strips all [X]HTML tags. For example:

```
{{ value|striptags }}
```

If `value` is `"<b>Joel</b> <button>is</button> a <span>slug</span>"`, the output will be `"Joel is a slug"`.

Again, these are just a few examples; see the *built-in filter reference* for the complete list.

You can also create your own custom template filters; see *Custom template tags and filters*.

See Also:

Django's admin interface can include a complete reference of all template tags and filters available for a given site. See *The Django admin documentation generator*.

3.5.4 Tags

Tags look like this: `{% tag %}`. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. `{% tag %} ... tag contents ... {% endtag %}`).

Django ships with about two dozen built-in template tags. You can read all about them in the *built-in tag reference*. To give you a taste of what's available, here are some of the more commonly used tags:

`for` Loop over each item in an array. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

`if` and `else` Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable.

You can also use filters and various operators in the `if` tag:

```
{% if athlete_list|length > 1 %}
   Team: {% for athlete in athlete_list %} ... {% endfor %}
{% else %}
   Athlete: {{ athlete_list.0.name }}
{% endif %}
```

While the above example works, be aware that most template filters return strings, so mathematical comparisons using filters will generally not work as you expect. `length` is an exception.

`block` and `extends` Set up template inheritance (see below), a powerful way of cutting down on "boilerplate" in templates.

Again, the above is only a selection of the whole list; see the *built-in tag reference* for the complete list.

You can also create your own custom template tags; see *Custom template tags and filters*.

See Also:

Django's admin interface can include a complete reference of all template tags and filters available for a given site. See *The Django admin documentation generator*.

3.5.5 Comments

To comment-out part of a line in a template, use the comment syntax: `{# #}`.

For example, this template would render as `'hello'`:

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

This syntax can only be used for single-line comments (no newlines are permitted between the `{#` and `#}` delimiters). If you need to comment out a multiline portion of the template, see the `comment` tag.

3.5.6 Template inheritance

The most powerful – and thus the most complex – part of Django's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

It's easiest to understand template inheritance by starting with an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
```

```
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content.

In this example, the `block` tag defines three blocks that child templates can fill in. All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this:

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

The `extends` tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent – in this case, "base.html".

At that point, the template engine will notice the three `block` tags in `base.html` and replace those blocks with the contents of the child template. Depending on the value of `blog_entries`, the output might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
```

```
</body>
</html>
```

Note that since the child template didn't define the `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a `base.html` template that holds the main look-and-feel of your site.
- Create a `base_SECTIONNAME.html` template for each "section" of your site. For example, `base_news.html`, `base_sports.html`. These templates all extend `base.html` and include section-specific styles/design.
- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.
- More `{% block %}` tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using `{{ block.super }}` will not be automatically escaped (see the next section), since it was already escaped, if necessary, in the parent template.
- For extra readability, you can optionally give a *name* to your `{% endblock %}` tag. For example:

  ```
  {% block content %}
  ...
  {% endblock content %}
  ```

  In larger templates, this technique helps you see which `{% block %}` tags are being closed.

Finally, note that you can't define multiple `block` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill – it also defines the content that fills the hole in the *parent*. If there were two similarly-named `block` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

3.5.7 Automatic HTML escaping

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. For example, consider this template fragment:

```
Hello, {{ name }}.
```

At first, this seems like a harmless way to display a user's name, but consider what would happen if the user entered his name as this:

```
<script>alert('hello')</script>
```

With this name value, the template would be rendered as:

```
Hello, <script>alert('hello')</script>
```

...which means the browser would pop-up a JavaScript alert box!

Similarly, what if the name contained a `'<'` symbol, like this?

```
<b>username
```

That would result in a rendered template like this:

```
Hello, <b>username
```

...which, in turn, would result in the remainder of the Web page being bolded!

Clearly, user-submitted data shouldn't be trusted blindly and inserted directly into your Web pages, because a malicious user could use this kind of hole to do potentially bad things. This type of security exploit is called a Cross Site Scripting (XSS) attack.

To avoid this problem, you have two options:

- One, you can make sure to run each untrusted variable through the `escape` filter (documented below), which converts potentially harmful HTML characters to unharmful ones. This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.
- Two, you can take advantage of Django's automatic HTML escaping. The remainder of this section describes how auto-escaping works.

By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- < is converted to `<`
- > is converted to `>`
- ' (single quote) is converted to `&#39;`
- " (double quote) is converted to `&quot;`
- & is converted to `&`

Again, we stress that this behavior is on by default. If you're using Django's template system, you're protected.

How to turn it off

If you don't want data to be auto-escaped, on a per-site, per-template level or per-variable level, you can turn it off in several ways.

Why would you want to turn it off? Because sometimes, template variables contain data that you *intend* to be rendered as raw HTML, in which case you don't want their contents to be escaped. For example, you might store a blob of HTML in your database and want to embed that directly into your template. Or, you might be using Django's template system to produce text that is *not* HTML – like an email message, for instance.

For individual variables

To disable auto-escaping for an individual variable, use the `safe` filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

Think of *safe* as shorthand for *safe from further escaping* or *can be safely interpreted as HTML*. In this example, if `data` contains `'<b>'`, the output will be:

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

For template blocks

To control auto-escaping for a template, wrap the template (or just a particular section of the template) in the `autoescape` tag, like so:

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

The `autoescape` tag takes either `on` or `off` as its argument. At times, you might want to force auto-escaping when it would otherwise be disabled. Here is an example template:

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
    {% autoescape on %}
        Auto-escaping applies again: {{ name }}
    {% endautoescape %}
{% endautoescape %}
```

The auto-escaping tag passes its effect onto templates that extend the current one as well as templates included via the `include` tag, just like all block tags. For example:

```
# base.html

{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}


# child.html

{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because auto-escaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the `greeting` variable contains the string `<b>Hello!</b>`:

```
<h1>This & that</h1>
<b>Hello!</b>
```

Notes

Generally, template authors don't need to worry about auto-escaping very much. Developers on the Python side (people writing views and custom filters) need to think about the cases in which data shouldn't be escaped, and mark data appropriately, so things Just Work in the template.

If you're creating a template that might be used in situations where you're not sure whether auto-escaping is enabled, then add an `escape` filter to any variable that needs escaping. When auto-escaping is on, there's no danger of the `escape` filter *double-escaping* data – the `escape` filter does not affect auto-escaped variables.

String literals and automatic escaping

As we mentioned earlier, filter arguments can be strings:

```
{{ data|default:"This is a string literal." }}
```

All string literals are inserted **without** any automatic escaping into the template – they act as if they were all passed through the `safe` filter. The reasoning behind this is that the template author is in control of what goes into the string literal, so they can make sure the text is correctly escaped when the template is written.

This means you would write

```
{{ data|default:"3 &lt; 2" }}
```

...rather than

```
{{ data|default:"3 < 2" }}  <-- Bad! Don't do this.
```

This doesn't affect what happens to data coming from the variable itself. The variable's contents are still automatically escaped, if necessary, because they're beyond the control of the template author.

3.5.8 Accessing method calls

Most method calls attached to objects are also available from within templates. This means that templates have access to much more than just class attributes (like field names) and variables passed in from views. For example, the Django ORM provides the *"entry_set"* syntax for finding a collection of objects related on a foreign key. Therefore, given a model called "comment" with a foreign key relationship to a model called "task" you can loop through all comments attached to a given task like this:

```
{% for comment in task.comment_set.all %}
    {{ comment }}
{% endfor %}
```

Similarly, *QuerySets* provide a `count()` method to count the number of objects they contain. Therefore, you can obtain a count of all comments related to the current task with:

```
{{ task.comment_set.all.count }}
```

And of course you can easily access methods you've explicitly defined on your own models:

```
# In model
class Task(models.Model):
    def foo(self):
        return "bar"

# In template
{{ task.foo }}
```

Because Django intentionally limits the amount of logic processing available in the template language, it is not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views, then passed to templates for display.

3.5.9 Custom tag and filter libraries

Certain applications provide custom tag and filter libraries. To access them in a template, use the `load` tag:

```
{% load comments %}

{% comment_form for blogs.entries entry.id with is_public yes %}
```

In the above, the `load` tag loads the `comments` tag library, which then makes the `comment_form` tag available for use. Consult the documentation area in your admin to find the list of custom libraries in your installation.

The `load` tag can take multiple library names, separated by spaces. Example:

```
{% load comments i18n %}
```

See *Custom template tags and filters* for information on writing your own custom template libraries.

Custom libraries and template inheritance

When you load a custom tag or filter library, the tags/filters are only made available to the current template – not any parent or child templates along the template-inheritance path.

For example, if a template `foo.html` has `{% load comments %}`, a child template (e.g., one that has `{% extends "foo.html" %}`) will *not* have access to the comments template tags and filters. The child template is responsible for its own `{% load comments %}`.

This is a feature for the sake of maintainability and sanity.

3.6 Class-based views

New in version 1.3: *Please see the release notes* A view is a callable which takes a request and returns a response. This can be more than just a function, and Django provides an example of some classes which can be used as views. These allow you to structure your views and reuse code by harnessing inheritance and mixins. There are also some generic views for simple tasks which we'll get to later, but you may want to design your own structure of reusable views which suits your use case. For full details, see the *class-based views reference documentation*.

3.6.1 Class-based generic views

Note: Prior to Django 1.3, generic views were implemented as functions. The function-based implementation has been removed in favor of the class-based approach described here.

Writing Web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but Web developers also experience this boredom at the view level.

Django's *generic views* were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.

We can recognize certain common tasks, like displaying a list of objects, and write code that displays a list of *any* object. Then the model in question can be passed as an extra argument to the URLconf.

Django ships with generic views to do the following:

- Display list and detail pages for a single object. If we were creating an application to manage conferences then a `TalkListView` and a `RegisteredUserListView` would be examples of list views. A single talk page is an example of what we call a "detail" view.
- Present date-based objects in year/month/day archive pages, associated detail, and "latest" pages.
- Allow users to create, update, and delete objects – with or without authorization.

Taken together, these views provide easy interfaces to perform the most common tasks developers encounter.

Extending generic views

There's no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is how to make generic views handle a wider array of situations.

This is one of the reasons generic views were redesigned for the 1.3 release - previously, they were just view functions with a bewildering array of options; now, rather than passing in a large amount of configuration in the URLconf, the recommended way to extend generic views is to subclass them, and override their attributes or methods.

That said, generic views will have a limit. If you find you're struggling to implement your view as a subclass of a generic view, then you may find it more effective to write just the code you need, using your own class-based or functional views.

More examples of generic views are available in some third party applications, or you could write your own as needed.

Generic views of objects

`TemplateView` certainly is useful, but Django's generic views really shine when it comes to presenting views of your database content. Because it's such a common task, Django comes with a handful of built-in generic views that make generating list and detail views of objects incredibly easy.

Let's start by looking at some examples of showing a list of objects or an individual object.

We'll be using these models:

```
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    class Meta:
        ordering = ["-name"]

    def __unicode__(self):
        return self.name

class Book(models.Model):
```

```
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

Now we need to define a view:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
```

Finally hook that view into your urls:

```
# urls.py
from django.conf.urls import patterns, url, include
from books.views import PublisherList

urlpatterns = patterns('',
    url(r'^publishers/$', PublisherList.as_view()),
)
```

That's all the Python code we need to write. We still need to write a template, however. We could explicitly tell the view which template to use by adding a `template_name` attribute to the view, but in the absence of an explicit template Django will infer one from the object's name. In this case, the inferred template will be `"books/publisher_list.html"` – the "books" part comes from the name of the app that defines the model, while the "publisher" bit is just the lowercased version of the model's name.

Note: Thus, when (for example) the `django.template.loaders.app_directories.Loader` template loader is enabled in `TEMPLATE_LOADERS`, a template location could be: /path/to/project/books/templates/books/publisher_list.html

---

This template will be rendered against a context containing a variable called `object_list` that contains all the publisher objects. A very simple template might look like the following:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publishers</h2>
    <ul>
        {% for publisher in object_list %}
            <li>{{ publisher.name }}</li>
        {% endfor %}
    </ul>
{% endblock %}
```

That's really all there is to it. All the cool features of generic views come from changing the attributes set on the generic view. The *generic views reference* documents all the generic views and their options in detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

Making "friendly" template contexts

You might have noticed that our sample publisher list template stores all the publishers in a variable named `object_list`. While this works just fine, it isn't all that "friendly" to template authors: they have to "just know"

that they're dealing with publishers here.

Well, if you're dealing with a model object, this is already done for you. When you are dealing with an object or queryset, Django is able to populate the context using the lower cased version of the model class' name. This is provided in addition to the default `object_list` entry, but contains exactly the same data, i.e. `publisher_list`.

If the this still isn't a good match, you can manually set the name of the context variable. The `context_object_name` attribute on a generic view specifies the context variable to use:

```
# views.py
from django.views.generic import ListView
from books.models import Publisher

class PublisherList(ListView):
    model = Publisher
    context_object_name = 'my_favourite_publishers'
```

Providing a useful `context_object_name` is always a good idea. Your coworkers who design templates will thank you.

Adding extra context

Often you simply need to present some extra information beyond that provided by the generic view. For example, think of showing a list of all the books on each publisher detail page. The `DetailView` generic view provides the publisher to the context, but how do we get additional information in that template.

However, there is; you can subclass `DetailView` and provide your own implementation of the `get_context_data` method. The default implementation of this that comes with `DetailView` simply adds in the object being displayed to the template, but you can override it to send more:

```
from django.views.generic import DetailView
from books.models import Publisher, Book

class PublisherDetail(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

Note: Generally, get_context_data will merge the context data of all parent classes with those of the current class. To preserve this behavior in your own classes where you want to alter the context, you should be sure to call get_context_data on the super class. When no two classes try to define the same key, this will give the expected results. However if any class attempts to override a key after parent classes have set it (after the call to super), any children of that class will also need to explictly set it after super if they want to be sure to override all parents. If you're having trouble, review the method resolution order of your view.

Viewing subsets of objects

Now let's take a closer look at the `model` argument we've been using all along. The `model` argument, which specifies the database model that the view will operate upon, is available on all the generic views that operate on a single object

or a collection of objects. However, the `model` argument is not the only way to specify the objects that the view will operate upon – you can also specify the list of objects using the `queryset` argument:

```
from django.views.generic import DetailView
from books.models import Publisher, Book

class PublisherDetail(DetailView):

    context_object_name = 'publisher'
    queryset = Publisher.objects.all()
```

Specifying `model = Publisher` is really just shorthand for saying `queryset = Publisher.objects.all()`. However, by using `queryset` to define a filtered list of objects you can be more specific about the objects that will be visible in the view (see *Making queries* for more information about `QuerySet` objects, and see the *class-based views reference* for the complete details).

To pick a simple example, we might want to order a list of books by publication date, with the most recent first:

```
from django.views.generic import ListView
from books.models import Book

class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

That's a pretty simple example, but it illustrates the idea nicely. Of course, you'll usually want to do more than just reorder objects. If you want to present a list of books by a particular publisher, you can use the same technique:

```
from django.views.generic import ListView
from books.models import Book

class AcmeBookList(ListView):

    context_object_name = 'book_list'
    queryset = Book.objects.filter(publisher__name='Acme Publishing')
    template_name = 'books/acme_list.html'
```

Notice that along with a filtered `queryset`, we're also using a custom template name. If we didn't, the generic view would use the same template as the "vanilla" object list, which might not be what we want.

Also notice that this isn't a very elegant way of doing publisher-specific books. If we want to add another publisher page, we'd need another handful of lines in the URLconf, and more than a few publishers would get unreasonable. We'll deal with this problem in the next section.

Note: If you get a 404 when requesting `/books/acme/`, check to ensure you actually have a Publisher with the name 'ACME Publishing'. Generic views have an `allow_empty` parameter for this case. See the *class-based-views reference* for more details.

Dynamic filtering

Another common need is to filter down the objects given in a list page by some key in the URL. Earlier we hard-coded the publisher's name in the URLconf, but what if we wanted to write a view that displayed all the books by some arbitrary publisher?

Handily, the `ListView` has a `get_queryset()` method we can override. Previously, it has just been returning the value of the `queryset` attribute, but now we can add more logic.

The key part to making this work is that when class-based views are called, various useful things are stored on `self`; as well as the request (`self.request`) this includes the positional (`self.args`) and name-based (`self.kwargs`) arguments captured according to the URLconf.

Here, we have a URLconf with a single captured group:

```
# urls.py
from books.views import PublisherBookList

urlpatterns = patterns('',
    (r'^books/([\w-]+)/$', PublisherBookList.as_view()),
)
```

Next, we'll write the `PublisherBookList` view itself:

```
# views.py
from django.shortcuts import get_object_or_404
from django.views.generic import ListView
from books.models import Book, Publisher

class PublisherBookList(ListView):

    template_name = 'books/books_by_publisher.html'

    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.args[0])
        return Book.objects.filter(publisher=self.publisher)
```

As you can see, it's quite easy to add more logic to the queryset selection; if we wanted, we could use `self.request.user` to filter using the current user, or other more complex logic.

We can also add the publisher into the context at the same time, so we can use it in the template:

```
# ...

def get_context_data(self, **kwargs):
    # Call the base implementation first to get a context
    context = super(PublisherBookList, self).get_context_data(**kwargs)
    # Add in the publisher
    context['publisher'] = self.publisher
    return context
```

Performing extra work

The last common pattern we'll look at involves doing some extra work before or after calling the generic view.

Imagine we had a `last_accessed` field on our `Author` object that we were using to keep track of the last time anybody looked at that author:

```
# models.py

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    name = models.CharField(max_length=200)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')
    last_accessed = models.DateTimeField()
```

The generic `DetailView` class, of course, wouldn't know anything about this field, but once again we could easily write a custom view to keep that field updated.

First, we'd need to add an author detail bit in the URLconf to point to a custom view:

```
from books.views import AuthorDetailView

urlpatterns = patterns('',
    #...
    url(r'^authors/(?P<pk>\d+)/$', AuthorDetailView.as_view(), name='author-detail'),
)
```

Then we'd write our new view – `get_object` is the method that retrieves the object – so we simply override it and wrap the call:

```
from django.views.generic import DetailView
from django.shortcuts import get_object_or_404
from django.utils import timezone
from books.models import Author

class AuthorDetailView(DetailView):

    queryset = Author.objects.all()

    def get_object(self):
        # Call the superclass
        object = super(AuthorDetailView, self).get_object()
        # Record the last accessed date
        object.last_accessed = timezone.now()
        object.save()
        # Return the object
        return object
```

Note: The URLconf here uses the named group `pk` - this name is the default name that `DetailView` uses to find the value of the primary key used to filter the queryset.

If you want to call the group something else, you can set `pk_url_kwarg` on the view. More details can be found in the reference for `DetailView`

3.6.2 Form handling with class-based views

Form processing generally has 3 paths:

- Initial GET (blank or prepopulated form)
- POST with invalid data (typically redisplay form with errors)
- POST with valid data (process the data and typically redirect)

Implementing this yourself often results in a lot of repeated boilerplate code (see *Using a form in a view*). To help avoid this, Django provides a collection of generic class-based views for form processing.

Basic Forms

Given a simple contact form:

```
# forms.py
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

The view can be constructed using a FormView:

```
# views.py
from myapp.forms import ContactForm
from django.views.generic.edit import FormView

class ContactView(FormView):
    template_name = 'contact.html'
    form_class = ContactForm
    success_url = '/thanks/'

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super(ContactView, self).form_valid(form)
```

Notes:

- FormView inherits `TemplateResponseMixin` so `template_name` can be used here
- The default implementation for `form_valid()` simply redirects to the `success_url`

Model Forms

Generic views really shine when working with models. These generic views will automatically create a `ModelForm`, so long as they can work out which model class to use:

- If the `model` attribute is given, that model class will be used
- If `get_object()` returns an object, the class of that object will be used
- If a `queryset` is given, the model for that queryset will be used

Model form views provide a `form_valid()` implementation that saves the model automatically. You can override this if you have any special requirements; see below for examples.

You don't even need to provide a attr:*success_url* for `CreateView` or `UpdateView` - they will use `get_absolute_url()` on the model object if available.

If you want to use a custom `ModelForm` (for instance to add extra validation) simply set `form_class` on your view.

Note: When specifying a custom form class, you must still specify the model, even though the `form_class` may be a `ModelForm`.

First we need to add `get_absolute_url()` to our `Author` class:

```
# models.py
from django import models
from django.core.urlresolvers import reverse

class Author(models.Model):
    name = models.CharField(max_length=200)

    def get_absolute_url(self):
        return reverse('author-detail', kwargs={'pk': self.pk})
```

Then we can use `CreateView` and friends to do the actual work. Notice how we're just configuring the generic class-based views here; we don't have to write any logic ourselves:

```
# views.py
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.core.urlresolvers import reverse_lazy
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author

class AuthorUpdate(UpdateView):
    model = Author

class AuthorDelete(DeleteView):
    model = Author
    success_url = reverse_lazy('author-list')
```

Note: We have to use `reverse_lazy()` here, not just `reverse` as the urls are not loaded when the file is imported.

Finally, we hook these new views into the URLconf:

```
# urls.py
from django.conf.urls import patterns, url
from myapp.views import AuthorCreate, AuthorUpdate, AuthorDelete

urlpatterns = patterns('',
    # ...
    url(r'author/add/$', AuthorCreate.as_view(), name='author_add'),
    url(r'author/(?P<pk>\d+)/$', AuthorUpdate.as_view(), name='author_update'),
    url(r'author/(?P<pk>\d+)/delete/$', AuthorDelete.as_view(), name='author_delete'),
)
```

Note: These views inherit `SingleObjectTemplateResponseMixin` which uses `template_name_prefix` to construct the `template_name` based on the model.

In this example:

- `CreateView` and `UpdateView` use `myapp/author_form.html`
- `DeleteView` uses `myapp/author_confirm_delete.html`

If you wish to have separate templates for `CreateView` and :class:1UpdateView', you can set either `template_name` or `template_name_suffix` on your view class.

Models and request.user

To track the user that created an object using a `CreateView`, you can use a custom `ModelForm` to do this. First, add the foreign key relation to the model:

```
# models.py
from django import models
from django.contrib.auth import User

class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User)

    # ...
```

Create a custom `ModelForm` in order to exclude the `created_by` field and prevent the user from editing it:

```
# forms.py
from django import forms
from myapp.models import Author

class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        exclude = ('created_by',)
```

In the view, use the custom `form_class` and override `form_valid()` to add the user:

```
# views.py
from django.views.generic.edit import CreateView
from myapp.models import Author
from myapp.forms import AuthorForm

class AuthorCreate(CreateView):
    form_class = AuthorForm
    model = Author

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super(AuthorCreate, self).form_valid(form)
```

Note that you'll need to *decorate this view* using `login_required()`, or alternatively handle unauthorised users in the `form_valid()`.

3.6.3 Using mixins with class-based views

New in version 1.3: *Please see the release notes*

Caution: This is an advanced topic. A working knowledge of *Django's class* exploring these techniques.

Django's built-in class-based views provide a lot of functionality, but some of it you may want to use separately. For instance, you may want to write a view that renders a template to make the HTTP response, but you can't use `TemplateView`; perhaps you need to render a template only on *POST*, with *GET* doing something else entirely. While you could use `TemplateResponse` directly, this will likely result in duplicate code.

For this reason, Django also provides a number of mixins that provide more discrete functionality. Template rendering, for instance, is encapsulated in the `TemplateResponseMixin`. The Django reference documentation contains *full documentation of all the mixins*.

Context and template responses

Two central mixins are provided that help in providing a consistent interface to working with templates in class-based views.

TemplateResponseMixin Every built in view which returns a `TemplateResponse` will call the `render_to_response()` method that `TemplateResponseMixin` provides. Most of the time this will be called for you (for instance, it is called by the `get()` method implemented by both `TemplateView` and `DetailView`); similarly, it's unlikely that you'll need to override it, although if you want your response to return something not rendered via a Django template then you'll want to do it. For an example of this, see the *JSONResponseMixin example*.

`render_to_response` itself calls `get_template_names()`, which by default will just look up `template_name` on the class-based view; two other mixins (`SingleObjectTemplateResponseMixin` and `MultipleObjectTemplateResponseMixin`) override this to provide more flexible defaults when dealing with actual objects.

New in version 1.5: *Please see the release notes*

ContextMixin Every built in view which needs context data, such as for rendering a template (including `TemplateResponseMixin` above), should call `get_context_data()` passing any data they want to ensure is in there as keyword arguments. `get_context_data` returns a dictionary; in `ContextMixin` it simply returns its keyword arguments, but it is common to override this to add more members to the dictionary.

Building up Django's generic class-based views

Let's look at how two of Django's generic class-based views are built out of mixins providing discrete functionality. We'll consider `DetailView`, which renders a "detail" view of an object, and `ListView`, which will render a list of objects, typically from a queryset, and optionally paginate them. This will introduce us to four mixins which between them provide useful functionality when working with either a single Django object, or multiple objects.

There are also mixins involved in the generic edit views (`FormView`, and the model-specific views `CreateView`, `UpdateView` and `DeleteView`), and in the date-based generic views. These are covered in the *mixin reference documentation*.

DetailView: working with a single Django object

To show the detail of an object, we basically need to do two things: we need to look up the object and then we need to make a `TemplateResponse` with a suitable template, and that object as context.

To get the object, `DetailView` relies on `SingleObjectMixin`, which provides a `get_object()` method that figures out the object based on the URL of the request (it looks for `pk` and `slug` keyword arguments as declared in the URLConf, and looks the object up either from the `model` attribute on the view, or the `queryset` attribute if that's provided). `SingleObjectMixin` also overrides `get_context_data()`, which is used across all Django's built in class-based views to supply context data for template renders.

To then make a `TemplateResponse`, `DetailView` uses `SingleObjectTemplateResponseMixin`, which extends `TemplateResponseMixin`, overriding `get_template_names()` as discussed above. It actually provides a fairly sophisticated set of options, but the main one that most people are going to use is `<app_label>/<object_name>_detail.html`. The `_detail` part can be changed by setting `template_name_suffix` on a subclass to something else. (For instance, the *generic edit views* use `_form` for create and update views, and `_confirm_delete` for delete views.)

ListView: working with many Django objects

Lists of objects follow roughly the same pattern: we need a (possibly paginated) list of objects, typically a `QuerySet`, and then we need to make a `TemplateResponse` with a suitable template using that list of objects.

To get the objects, `ListView` uses `MultipleObjectMixin`, which provides both `get_queryset()` and `paginate_queryset()`. Unlike with `SingleObjectMixin`, there's no need to key off parts of the URL to figure out the queryset to work with, so the default just uses the `queryset` or `model` attribute on the view class. A common reason to override `get_queryset()` here would be to dynamically vary the objects, such as depending on the current user or to exclude posts in the future for a blog.

`MultipleObjectMixin` also overrides `get_context_data()` to include appropriate context variables for pagination (providing dummies if pagination is disabled). It relies on `object_list` being passed in as a keyword argument, which `ListView` arranges for it.

To make a `TemplateResponse`, `ListView` then uses `MultipleObjectTemplateResponseMixin`; as with `SingleObjectTemplateResponseMixin` above, this overrides `get_template_names()` to provide a range of options, with the most commonly-used being `<app_label>/<object_name>_list.html`, with the `_list` part again being taken from the `template_name_suffix` attribute. (The date based generic views use suffixes such as `_archive`, `_archive_year` and so on to use different templates for the various specialised date-based list views.)

Using Django's class-based view mixins

Now we've seen how Django's generic class-based views use the provided mixins, let's look at other ways we can combine them. Of course we're still going to be combining them with either built-in class-based views, or other generic class-based views, but there are a range of rarer problems you can solve than are provided for by Django out of the box.

Warning: Not all mixins can be used together, and not all generic class based views can be used with all other mixins. Here we present a few examples that do work; if you want to bring together other functionality then you'll have to consider interactions between attributes and methods that overlap between the different classes you're using, and how method resolution order will affect which versions of the methods will be called in what order.
The reference documentation for Django's *class-based views* and *class-based view mixins* will help you in understanding which attributes and methods are likely to cause conflict between different classes and mixins.
If in doubt, it's often better to back off and base your work on `View` or `TemplateView`, perhaps with `SimpleObjectMixin` and `MultipleObjectMixin`. Although you will probably end up writing more code, it is more likely to be clearly understandable to someone else coming to it later, and with fewer interactions to worry about you will save yourself some thinking. (Of course, you can always dip into Django's implementation of the generic class based views for inspiration on how to tackle problems.)

Using SingleObjectMixin with View

If we want to write a simple class-based view that responds only to `POST`, we'll subclass `View` and write a `post()` method in the subclass. However if we want our processing to work on a particular object, identified from the URL, we'll want the functionality provided by `SingleObjectMixin`.

We'll demonstrate this with the publisher modelling we used in the *generic class-based views introduction*.

```
# views.py
from django.http import HttpResponseForbidden, HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.views.generic import View
from django.views.generic.detail import SingleObjectMixin
```

```
from books.models import Author

class RecordInterest(View, SingleObjectMixin):
    """Records the current user's interest in an author."""
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            return HttpResponseForbidden()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(reverse('author-detail', kwargs={'pk': self.object.pk}))
```

In practice you'd probably want to record the interest in a key-value store rather than in a relational database, so we've left that bit out. The only bit of the view that needs to worry about using `SingleObjectMixin` is where we want to look up the author we're interested in, which it just does with a simple call to `self.get_object()`. Everything else is taken care of for us by the mixin.

We can hook this into our URLs easily enough:

```
# urls.py
from books.views import RecordInterest

urlpatterns = patterns('',
    #...
    url(r'^author/(?P<pk>\d+)/interest/$', RecordInterest.as_view(), name='author-interest'),
)
```

Note the `pk` named group, which `get_object()` uses to look up the `Author` instance. You could also use a slug, or any of the other features of `SingleObjectMixin`.

Using SingleObjectMixin with ListView

`ListView` provides built-in pagination, but you might want to paginate a list of objects that are all linked (by a foreign key) to another object. In our publishing example, you might want to paginate through all the books by a particular publisher.

One way to do this is to combine `ListView` with `SingleObjectMixin`, so that the queryset for the paginated list of books can hang off the publisher found as the single object. In order to do this, we need to have two different querysets:

Publisher queryset for use in get_object We'll set that up directly when we call `get_object()`.

Book queryset for use by ListView We'll figure that out ourselves in `get_queryset()` so we can take into account the Publisher we're looking at.

Note: We have to think carefully about `get_context_data()`. Since both `SingleObjectMixin` and `ListView` will put things in the context data under the value of `context_object_name` if it's set, we'll instead explictly ensure the Publisher is in the context data. `ListView` will add in the suitable `page_obj` and `paginator` for us providing we remember to call `super()`.

Now we can write a new `PublisherDetail`:

```
from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin
from books.models import Publisher

class PublisherDetail(SingleObjectMixin, ListView):
    paginate_by = 2
    template_name = "books/publisher_detail.html"

    def get_context_data(self, **kwargs):
        kwargs['publisher'] = self.object
        return super(PublisherDetail, self).get_context_data(**kwargs)

    def get_queryset(self):
        self.object = self.get_object(Publisher.objects.all())
        return self.object.book_set.all()
```

Notice how we set `self.object` within `get_queryset()` so we can use it again later in `get_context_data()`. If you don't set `template_name`, the template will default to the normal `ListView` choice, which in this case would be `"books/book_list.html"` because it's a list of books; `ListView` knows nothing about `SingleObjectMixin`, so it doesn't have any clue this view is anything to do with a Publisher.

The `paginate_by` is deliberately small in the example so you don't have to create lots of books to see the pagination working! Here's the template you'd want to use:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publisher {{ publisher.name }}</h2>

    <ol>
      {% for book in page_obj %}
        <li>{{ book.title }}</li>
      {% endfor %}
    </ol>

    <div class="pagination">
        <span class="step-links">
            {% if page_obj.has_previous %}
                <a href="?page={{ page_obj.previous_page_number }}">previous</a>
            {% endif %}

            <span class="current">
                Page {{ page_obj.number }} of {{ paginator.num_pages }}.
            </span>

            {% if page_obj.has_next %}
                <a href="?page={{ page_obj.next_page_number }}">next</a>
            {% endif %}
        </span>
    </div>
{% endblock %}
```

Avoid anything more complex

Generally you can use `TemplateResponseMixin` and `SingleObjectMixin` when you need their functionality. As shown above, with a bit of care you can even combine `SingleObjectMixin` with `ListView`. However things get increasingly complex as you try to do so, and a good rule of thumb is:

Hint: Each of your views should use only mixins or views from one of the groups of generic class-based views: *detail, list*, *editing* and date. For example it's fine to combine `TemplateView` (built in view) with `MultipleObjectMixin` (generic list), but you're likely to have problems combining `SingleObjectMixin` (generic detail) with `MultipleObjectMixin` (generic list).

To show what happens when you try to get more sophisticated, we show an example that sacrifices readability and maintainability when there is a simpler solution. First, let's look at a naive attempt to combine `DetailView` with `FormMixin` to enable use to `POST` a Django `Form` to the same URL as we're displaying an object using `DetailView`.

Using FormMixin with DetailView

Think back to our earlier example of using `View` and `SingleObjectMixin` together. We were recording a user's interest in a particular author; say now that we want to let them leave a message saying why they like them. Again, let's assume we're not going to store this in a relational database but instead in something more esoteric that we won't worry about here.

At this point it's natural to reach for a `Form` to encapsulate the information sent from the user's browser to Django. Say also that we're heavily invested in REST, so we want to use the same URL for displaying the author as for capturing the message from the user. Let's rewrite our `AuthorDetailView` to do that.

We'll keep the `GET` handling from `DetailView`, although we'll have to add a `Form` into the context data so we can render it in the template. We'll also want to pull in form processing from `FormMixin`, and write a bit of code so that on `POST` the form gets called appropriately.

Note: We use `FormMixin` and implement `post()` ourselves rather than try to mix `DetailView` with `FormView` (which provides a suitable `post()` already) because both of the views implement `get()`, and things would get much more confusing.

Our new `AuthorDetail` looks like this:

```
# CAUTION: you almost certainly do not want to do this.
# It is provided as part of a discussion of problems you can
# run into when combining different generic class-based view
# functionality that is not designed to be used together.

from django import forms
from django.http import HttpResponseForbidden
from django.core.urlresolvers import reverse
from django.views.generic import DetailView
from django.views.generic.edit import FormMixin

class AuthorInterestForm(forms.Form):
    message = forms.CharField()

class AuthorDetail(DetailView, FormMixin):
    model = Author
    form_class = AuthorInterestForm

    def get_success_url(self):
        return reverse(
            'author-detail',
            kwargs = {'pk': self.object.pk},
        )
```

```
    def get_context_data(self, **kwargs):
        form_class = self.get_form_class()
        form = self.get_form(form_class)
        context = {
            'form': form
        }
        context.update(kwargs)
        return super(AuthorDetail, self).get_context_data(**context)

    def post(self, request, *args, **kwargs):
        form_class = self.get_form_class()
        form = self.get_form(form_class)
        if form.is_valid():
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

    def form_valid(self, form):
        if not self.request.user.is_authenticated():
            return HttpResponseForbidden()
        self.object = self.get_object()
        # record the interest using the message in form.cleaned_data
        return super(AuthorDetail, self).form_valid(form)
```

`get_success_url()` is just providing somewhere to redirect to, which gets used in the default implementation of `form_valid()`. We have to provide our own `post()` as noted earlier, and override `get_context_data()` to make the `Form` available in the context data.

A better solution

It should be obvious that the number of subtle interactions between `FormMixin` and `DetailView` is already testing our ability to manage things. It's unlikely you'd want to write this kind of class yourself.

In this case, it would be fairly easy to just write the `post()` method yourself, keeping `DetailView` as the only generic functionality, although writing `Form` handling code involves a lot of duplication.

Alternatively, it would still be easier than the above approach to have a separate view for processing the form, which could use `FormView` distinct from `DetailView` without concerns.

An alternative better solution

What we're really trying to do here is to use two different class based views from the same URL. So why not do just that? We have a very clear division here: `GET` requests should get the `DetailView` (with the `Form` added to the context data), and `POST` requests should get the `FormView`. Let's set up those views first.

The `AuthorDisplay` view is almost the same as *when we first introduced AuthorDetail*; we have to write our own `get_context_data()` to make the `AuthorInterestForm` available to the template. We'll skip the `get_object()` override from before for clarity.

```
from django.views.generic import DetailView
from django import forms
from books.models import Author

class AuthorInterestForm(forms.Form):
    message = forms.CharField()
```

```
class AuthorDisplay(DetailView):

    queryset = Author.objects.all()

    def get_context_data(self, **kwargs):
        context = {
            'form': AuthorInterestForm(),
        }
        context.update(kwargs)
        return super(AuthorDisplay, self).get_context_data(**context)
```

Then the `AuthorInterest` is a simple `FormView`, but we have to bring in `SingleObjectMixin` so we can find the author we're talking about, and we have to remember to set `template_name` to ensure that form errors will render the same template as `AuthorDisplay` is using on `GET`.

```
from django.views.generic import FormView
from django.views.generic.detail import SingleObjectMixin

class AuthorInterest(FormView, SingleObjectMixin):
    template_name = 'books/author_detail.html'
    form_class = AuthorInterestForm
    model = Author

    def get_context_data(self, **kwargs):
        context = {
            'object': self.get_object(),
        }
        return super(AuthorInterest, self).get_context_data(**context)

    def get_success_url(self):
        return reverse(
            'author-detail',
            kwargs = {'pk': self.object.pk},
        )

    def form_valid(self, form):
        if not self.request.user.is_authenticated():
            return HttpResponseForbidden()
        self.object = self.get_object()
        # record the interest using the message in form.cleaned_data
        return super(AuthorInterest, self).form_valid(form)
```

Finally we bring this together in a new `AuthorDetail` view. We already know that calling `as_view()` on a class-based view gives us something that behaves exactly like a function based view, so we can do that at the point we choose between the two subviews.

You can of course pass through keyword arguments to `as_view()` in the same way you would in your URLconf, such as if you wanted the `AuthorInterest` behaviour to also appear at another URL but using a different template.

```
from django.views.generic import View

class AuthorDetail(View):

    def get(self, request, *args, **kwargs):
        view = AuthorDisplay.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = AuthorInterest.as_view()
```

```
        return view(request, *args, **kwargs)
```

This approach can also be used with any other generic class-based views or your own class-based views inheriting directly from `View` or `TemplateView`, as it keeps the different views as separate as possible.

3.6.4 Basic examples

Django provides base view classes which will suit a wide range of applications. All views inherit from the `View` class, which handles linking the view in to the URLs, HTTP method dispatching and other simple features. `RedirectView` is for a simple HTTP redirect, and `TemplateView` extends the base class to make it also render a template.

3.6.5 Simple usage

Class-based generic views (and any class-based views that inherit from the base classes Django provides) can be configured in two ways: subclassing, or passing in arguments directly in the URLconf.

When you subclass a class-based view, you can override attributes (such as the `template_name`) or methods (such as `get_context_data`) in your subclass to provide new values or methods. Consider, for example, a view that just displays one template, `about.html`. Django has a generic view to do this - `TemplateView` - so we can just subclass it, and override the template name:

```
# some_app/views.py
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
```

Then, we just need to add this new view into our URLconf. As the class-based views themselves are classes, we point the URL to the `as_view` class method instead, which is the entry point for class-based views:

```
# urls.py
from django.conf.urls import patterns, url, include
from some_app.views import AboutView

urlpatterns = patterns('',
    (r'^about/', AboutView.as_view()),
)
```

Alternatively, if you're only changing a few simple attributes on a class-based view, you can simply pass the new attributes into the `as_view` method call itself:

```
from django.conf.urls import patterns, url, include
from django.views.generic import TemplateView

urlpatterns = patterns('',
    (r'^about/', TemplateView.as_view(template_name="about.html")),
)
```

A similar overriding pattern can be used for the `url` attribute on `RedirectView`.

More than just HTML

Where class based views shine is when you want to do the same thing many times. Suppose you're writing an API, and every view should return JSON instead of rendered HTML.

We can use create a mixin class to use in all of our views, handling the conversion to JSON once.

For example, a simple JSON mixin might look something like this:

```
import json
from django.http import HttpResponse

class JSONResponseMixin(object):
    """
    A mixin that can be used to render a JSON response.
    """
    response_class = HttpResponse

    def render_to_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        response_kwargs['content_type'] = 'application/json'
        return self.response_class(
            self.convert_context_to_json(context),
            **response_kwargs
        )

    def convert_context_to_json(self, context):
        "Convert the context dictionary into a JSON object"
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return json.dumps(context)
```

Now we mix this into the base view:

```
from django.views.generic import View

class JSONView(JSONResponseMixin, View):
    pass
```

Equally we could use our mixin with one of the generic views. We can make our own version of `DetailView` by mixing `JSONResponseMixin` with the `BaseDetailView` – (the `DetailView` before template rendering behavior has been mixed in):

```
class JSONDetailView(JSONResponseMixin, BaseDetailView):
    pass
```

This view can then be deployed in the same way as any other `DetailView`, with exactly the same behavior – except for the format of the response.

If you want to be really adventurous, you could even mix a `DetailView` subclass that is able to return *both* HTML and JSON content, depending on some property of the HTTP request, such as a query argument or a HTTP header. Just mix in both the `JSONResponseMixin` and a `SingleObjectTemplateResponseMixin`, and override the implementation of `render_to_response()` to defer to the appropriate subclass depending on the type of response that the user requested:

```
class HybridDetailView(JSONResponseMixin, SingleObjectTemplateResponseMixin, BaseDetailView):
    def render_to_response(self, context):
        # Look for a 'format=json' GET argument
        if self.request.GET.get('format','html') == 'json':
            return JSONResponseMixin.render_to_response(self, context)
        else:
            return SingleObjectTemplateResponseMixin.render_to_response(self, context)
```

Because of the way that Python resolves method overloading, the local `render_to_response()` implementation will override the versions provided by `JSONResponseMixin` and `SingleObjectTemplateResponseMixin`.

For more information on how to use the built in generic views, consult the next topic on *generic class based views*.

3.6.6 Decorating class-based views

The extension of class-based views isn't limited to using mixins. You can use also use decorators.

Decorating in URLconf

The simplest way of decorating class-based views is to decorate the result of the `as_view()` method. The easiest place to do this is in the URLconf where you deploy your view:

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.generic import TemplateView

from .views import VoteView

urlpatterns = patterns('',
    (r'^about/', login_required(TemplateView.as_view(template_name="secret.html"))),
    (r'^vote/', permission_required('polls.can_vote')(VoteView.as_view())),
)
```

This approach applies the decorator on a per-instance basis. If you want every instance of a view to be decorated, you need to take a different approach.

Decorating the class

To decorate every instance of a class-based view, you need to decorate the class definition itself. To do this you apply the decorator to the `dispatch()` method of the class.

A method on a class isn't quite the same as a standalone function, so you can't just apply a function decorator to the method – you need to transform it into a method decorator first. The `method_decorator` decorator transforms a function decorator into a method decorator so that it can be used on an instance method. For example:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

class ProtectedView(TemplateView):
    template_name = 'secret.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super(ProtectedView, self).dispatch(*args, **kwargs)
```

In this example, every instance of `ProtectedView` will have login protection.

Note: `method_decorator` passes `*args` and `**kwargs` as parameters to the decorated method on the class. If your method does not accept a compatible set of parameters it will raise a `TypeError` exception.

3.7 Managing files

This document describes Django's file access APIs.

By default, Django stores files locally, using the `MEDIA_ROOT` and `MEDIA_URL` settings. The examples below assume that you're using these defaults.

However, Django provides ways to write custom file storage systems that allow you to completely customize where and how Django stores files. The second half of this document describes how these storage systems work.

3.7.1 Using files in models

When you use a `FileField` or `ImageField`, Django provides a set of APIs you can use to deal with that file.

Consider the following model, using an `ImageField` to store a photo:

```
class Car(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=5, decimal_places=2)
    photo = models.ImageField(upload_to='cars')
```

Any `Car` instance will have a `photo` attribute that you can use to get at the details of the attached photo:

```
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo
<ImageFieldFile: chevy.jpg>
>>> car.photo.name
u'cars/chevy.jpg'
>>> car.photo.path
u'/media/cars/chevy.jpg'
>>> car.photo.url
u'http://media.example.com/cars/chevy.jpg'
```

This object – `car.photo` in the example – is a `File` object, which means it has all the methods and attributes described below.

Note: The file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until after the model has been saved.

3.7.2 The `File` object

Internally, Django uses a `django.core.files.File` instance any time it needs to represent a file. This object is a thin wrapper around Python's built-in file object with some Django-specific additions.

Most of the time you'll simply use a `File` that Django's given you (i.e. a file attached to a model as above, or perhaps an uploaded file).

If you need to construct a `File` yourself, the easiest way is to create one using a Python built-in `file` object:

```
>>> from django.core.files import File

# Create a Python file object using open()
>>> f = open('/tmp/hello.world', 'w')
>>> myfile = File(f)
```

Now you can use any of the documented attributes and methods of the `File` class.

Be aware that files created in this way are not automatically closed. The following approach may be used to close files automatically:

```
>>> from django.core.files import File

# Create a Python file object using open() and the with statement
>>> with open('/tmp/hello.world', 'w') as f:
>>>     myfile = File(f)
>>>     for line in myfile:
>>>         print line
>>> myfile.closed
True
>>> f.closed
True
```

Closing files is especially important when accessing file fields in a loop over a large number of objects:: If files are not manually closed after accessing them, the risk of running out of file descriptors may arise. This may lead to the following error:

> IOError: [Errno 24] Too many open files

3.7.3 File storage

Behind the scenes, Django delegates decisions about how and where to store files to a file storage system. This is the object that actually understands things like file systems, opening and reading files, etc.

Django's default file storage is given by the `DEFAULT_FILE_STORAGE` setting; if you don't explicitly provide a storage system, this is the one that will be used.

See below for details of the built-in default file storage system, and see *Writing a custom storage system* for information on writing your own file storage system.

Storage objects

Though most of the time you'll want to use a `File` object (which delegates to the proper storage for that file), you can use file storage systems directly. You can create an instance of some custom file storage class, or – often more useful – you can use the global default storage system:

```
>>> from django.core.files.storage import default_storage
>>> from django.core.files.base import ContentFile

>>> path = default_storage.save('/path/to/file', ContentFile('new content'))
>>> path
u'/path/to/file'

>>> default_storage.size(path)
11
>>> default_storage.open(path).read()
'new content'

>>> default_storage.delete(path)
>>> default_storage.exists(path)
False
```

See *File storage API* for the file storage API.

The built-in filesystem storage class

Django ships with a built-in `FileSystemStorage` class (defined in `django.core.files.storage`) which implements basic local filesystem file storage. Its initializer takes two arguments:

Argument	Description
`location`	Optional. Absolute path to the directory that will hold the files. If omitted, it will be set to the value of your `MEDIA_ROOT` setting.
`base_url`	Optional. URL that serves the files stored at this location. If omitted, it will default to the value of your `MEDIA_URL` setting.

For example, the following code will store uploaded files under `/media/photos` regardless of what your `MEDIA_ROOT` setting is:

```
from django.db import models
from django.core.files.storage import FileSystemStorage

fs = FileSystemStorage(location='/media/photos')

class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)
```

Custom storage systems work the same way: you can pass them in as the `storage` argument to a `FileField`.

3.8 Testing Django applications

Automated testing is an extremely useful bug-killing tool for the modern Web developer. You can use a collection of tests – a **test suite** – to solve, or avoid, a number of problems:

- When you're writing new code, you can use tests to validate your code works as expected.
- When you're refactoring or modifying old code, you can use tests to ensure your changes haven't affected your application's behavior unexpectedly.

Testing a Web application is a complex task, because a Web application is made of several layers of logic – from HTTP-level request handling, to form validation and processing, to template rendering. With Django's test-execution framework and assorted utilities, you can simulate requests, insert test data, inspect your application's output and generally verify your code is doing what it should be doing.

The best part is, it's really easy.

This document is split into two primary sections. First, we explain how to write tests with Django. Then, we explain how to run them.

3.8.1 Writing tests

There are two primary ways to write tests with Django, corresponding to the two test frameworks that ship in the Python standard library. The two frameworks are:

- **Unit tests** – tests that are expressed as methods on a Python class that subclasses `unittest.TestCase` or Django's customized `TestCase`. For example:

  ```
  import unittest

  class MyFuncTestCase(unittest.TestCase):
  ```

```
    def testBasic(self):
        a = ['larry', 'curly', 'moe']
        self.assertEqual(my_func(a, 0), 'larry')
        self.assertEqual(my_func(a, 1), 'curly')
```

- **Doctests** – tests that are embedded in your functions' docstrings and are written in a way that emulates a session of the Python interactive interpreter. For example:

```
def my_func(a_list, idx):
    """
    >>> a = ['larry', 'curly', 'moe']
    >>> my_func(a, 0)
    'larry'
    >>> my_func(a, 1)
    'curly'
    """
    return a_list[idx]
```

We'll discuss choosing the appropriate test framework later, however, most experienced developers prefer unit tests. You can also use any *other* Python test framework, as we'll explain in a bit.

Writing unit tests

Django's unit tests use a Python standard library module: `unittest`. This module defines tests in class-based approach.

---

unittest2 Changed in version 1.3: *Please see the release notes* Python 2.7 introduced some major changes to the unittest library, adding some extremely useful features. To ensure that every Django project can benefit from these new features, Django ships with a copy of unittest2, a copy of the Python 2.7 unittest library, backported for Python 2.5 compatibility.

To access this library, Django provides the `django.utils.unittest` module alias. If you are using Python 2.7, or you have installed unittest2 locally, Django will map the alias to the installed version of the unittest library. Otherwise, Django will use its own bundled version of unittest2.

To use this alias, simply use:

```
from django.utils import unittest
```

wherever you would have historically used:

```
import unittest
```

If you want to continue to use the base unittest library, you can – you just won't get any of the nice new unittest2 features.

For a given Django application, the test runner looks for unit tests in two places:

- The `models.py` file. The test runner looks for any subclass of `unittest.TestCase` in this module.
- A file called `tests.py` in the application directory – i.e., the directory that holds `models.py`. Again, the test runner looks for any subclass of `unittest.TestCase` in this module.

Here is an example `unittest.TestCase` subclass:

```
from django.utils import unittest
from myapp.models import Animal
```

```
class AnimalTestCase(unittest.TestCase):
    def setUp(self):
        self.lion = Animal.objects.create(name="lion", sound="roar")
        self.cat = Animal.objects.create(name="cat", sound="meow")

    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        self.assertEqual(self.lion.speak(), 'The lion says "roar"')
        self.assertEqual(self.cat.speak(), 'The cat says "meow"')
```

When you *run your tests*, the default behavior of the test utility is to find all the test cases (that is, subclasses of `unittest.TestCase`) in `models.py` and `tests.py`, automatically build a test suite out of those test cases, and run that suite.

There is a second way to define the test suite for a module: if you define a function called `suite()` in either `models.py` or `tests.py`, the Django test runner will use that function to construct the test suite for that module. This follows the suggested organization for unit tests. See the Python documentation for more details on how to construct a complex test suite.

For more details about `unittest`, see the Python documentation.

Writing doctests

Doctests use Python's standard `doctest` module, which searches your docstrings for statements that resemble a session of the Python interactive interpreter. A full explanation of how `doctest` works is out of the scope of this document; read Python's official documentation for the details.

What's a docstring?

A good explanation of docstrings (and some guidelines for using them effectively) can be found in **PEP 257**:

> A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

For example, this function has a docstring that describes what it does:

```
def add_two(num):
    "Return the result of adding two to the provided number."
    return num + 2
```

Because tests often make great documentation, putting tests directly in your docstrings is an effective way to document *and* test your code.

As with unit tests, for a given Django application, the test runner looks for doctests in two places:

- The `models.py` file. You can define module-level doctests and/or a doctest for individual models. It's common practice to put application-level doctests in the module docstring and model-level doctests in the model docstrings.
- A file called `tests.py` in the application directory – i.e., the directory that holds `models.py`. This file is a hook for any and all doctests you want to write that aren't necessarily related to models.

This example doctest is equivalent to the example given in the unittest section above:

```
# models.py

from django.db import models
```

```
class Animal(models.Model):
    """
    An animal that knows how to make noise

    # Create some animals
    >>> lion = Animal.objects.create(name="lion", sound="roar")
    >>> cat = Animal.objects.create(name="cat", sound="meow")

    # Make 'em speak
    >>> lion.speak()
    'The lion says "roar"'
    >>> cat.speak()
    'The cat says "meow"'
    """
    name = models.CharField(max_length=20)
    sound = models.CharField(max_length=20)

    def speak(self):
        return 'The %s says "%s"' % (self.name, self.sound)
```

When you *run your tests*, the test runner will find this docstring, notice that portions of it look like an interactive Python session, and execute those lines while checking that the results match.

In the case of model tests, note that the test runner takes care of creating its own test database. That is, any test that accesses a database – by creating and saving model instances, for example – will not affect your production database. However, the database is not refreshed between doctests, so if your doctest requires a certain state you should consider flushing the database or loading a fixture. (See the section on fixtures, below, for more on this.) Note that to use this feature, the database user Django is connecting as must have `CREATE DATABASE` rights.

For more details about `doctest`, see the Python documentation.

Which should I use?

Because Django supports both of the standard Python test frameworks, it's up to you and your tastes to decide which one to use. You can even decide to use *both*.

For developers new to testing, however, this choice can seem confusing. Here, then, are a few key differences to help you decide which approach is right for you:

- If you've been using Python for a while, `doctest` will probably feel more "pythonic". It's designed to make writing tests as easy as possible, so it requires no overhead of writing classes or methods. You simply put tests in docstrings. This has the added advantage of serving as documentation (and correct documentation, at that!). However, while doctests are good for some simple example code, they are not very good if you want to produce either high quality, comprehensive tests or high quality documentation. Test failures are often difficult to debug as it can be unclear exactly why the test failed. Thus, doctests should generally be avoided and used primarily for documentation examples only.
- The `unittest` framework will probably feel very familiar to developers coming from Java. `unittest` is inspired by Java's JUnit, so you'll feel at home with this method if you've used JUnit or any test framework inspired by JUnit.
- If you need to write a bunch of tests that share similar code, then you'll appreciate the `unittest` framework's organization around classes and methods. This makes it easy to abstract common tasks into common methods. The framework also supports explicit setup and/or cleanup routines, which give you a high level of control over the environment in which your test cases are run.
- If you're writing tests for Django itself, you should use `unittest`.

3.8.2 Running tests

Once you've written tests, run them using the `test` command of your project's `manage.py` utility:

```
$ ./manage.py test
```

By default, this will run every test in every application in `INSTALLED_APPS`. If you only want to run tests for a particular application, add the application name to the command line. For example, if your `INSTALLED_APPS` contains `'myproject.polls'` and `'myproject.animals'`, you can run the `myproject.animals` unit tests alone with this command:

```
$ ./manage.py test animals
```

Note that we used `animals`, not `myproject.animals`.

You can be even *more* specific by naming an individual test case. To run a single test case in an application (for example, the `AnimalTestCase` described in the "Writing unit tests" section), add the name of the test case to the label on the command line:

```
$ ./manage.py test animals.AnimalTestCase
```

And it gets even more granular than that! To run a *single* test method inside a test case, add the name of the test method to the label:

```
$ ./manage.py test animals.AnimalTestCase.test_animals_can_speak
```

You can use the same rules if you're using doctests. Django will use the test label as a path to the test method or class that you want to run. If your `models.py` or `tests.py` has a function with a doctest, or class with a class-level doctest, you can invoke that test by appending the name of the test method or class to the label:

```
$ ./manage.py test animals.classify
```

If you want to run the doctest for a specific method in a class, add the name of the method to the label:

```
$ ./manage.py test animals.Classifier.run
```

If you're using a `__test__` dictionary to specify doctests for a module, Django will use the label as a key in the `__test__` dictionary for defined in `models.py` and `tests.py`.

If you press `Ctrl-C` while the tests are running, the test runner will wait for the currently running test to complete and then exit gracefully. During a graceful exit the test runner will output details of any test failures, report on how many tests were run and how many errors and failures were encountered, and destroy any test databases as usual. Thus pressing `Ctrl-C` can be very useful if you forget to pass the `--failfast` option, notice that some tests are unexpectedly failing, and want to get details on the failures without waiting for the full test run to complete.

If you do not want to wait for the currently running test to finish, you can press `Ctrl-C` a second time and the test run will halt immediately, but not gracefully. No details of the tests run before the interruption will be reported, and any test databases created by the run will not be destroyed.

Test with warnings enabled

It's a good idea to run your tests with Python warnings enabled: `python -Wall manage.py test`. The `-Wall` flag tells Python to display deprecation warnings. Django, like many other Python libraries, uses these warnings to flag when features are going away. It also might flag areas in your code that aren't strictly wrong but could benefit from a better implementation.

Running tests outside the test runner

If you want to run tests outside of `./manage.py test` – for example, from a shell prompt – you will need to set up the test environment first. Django provides a convenience method to do this:

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()
```

This convenience method sets up the test database, and puts other Django features into modes that allow for repeatable testing.

The call to `setup_test_environment()` is made automatically as part of the setup of *./manage.py test*. You only need to manually invoke this method if you're not using running your tests via Django's test runner.

The test database

Tests that require a database (namely, model tests) will not use your "real" (production) database. Separate, blank databases are created for the tests.

Regardless of whether the tests pass or fail, the test databases are destroyed when all the tests have been executed.

By default the test databases get their names by prepending `test_` to the value of the `NAME` settings for the databases defined in `DATABASES`. When using the SQLite database engine the tests will by default use an in-memory database (i.e., the database will be created in memory, bypassing the filesystem entirely!). If you want to use a different database name, specify `TEST_NAME` in the dictionary for any given database in `DATABASES`.

Aside from using a separate database, the test runner will otherwise use all of the same database settings you have in your settings file: `ENGINE`, `USER`, `HOST`, etc. The test database is created by the user specified by `USER`, so you'll need to make sure that the given user account has sufficient privileges to create a new database on the system.

For fine-grained control over the character encoding of your test database, use the `TEST_CHARSET` option. If you're using MySQL, you can also use the `TEST_COLLATION` option to control the particular collation used by the test database. See the *settings documentation* for details of these advanced settings.

Testing master/slave configurations

If you're testing a multiple database configuration with master/slave replication, this strategy of creating test databases poses a problem. When the test databases are created, there won't be any replication, and as a result, data created on the master won't be seen on the slave.

To compensate for this, Django allows you to define that a database is a *test mirror*. Consider the following (simplified) example database configuration:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'myproject',
        'HOST': 'dbmaster',
         # ... plus some other settings
    },
    'slave': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'myproject',
        'HOST': 'dbslave',
        'TEST_MIRROR': 'default'
        # ... plus some other settings
```

```
    }
}
```

In this setup, we have two database servers: `dbmaster`, described by the database alias `default`, and `dbslave` described by the alias `slave`. As you might expect, `dbslave` has been configured by the database administrator as a read slave of `dbmaster`, so in normal activity, any write to `default` will appear on `slave`.

If Django created two independent test databases, this would break any tests that expected replication to occur. However, the `slave` database has been configured as a test mirror (using the `TEST_MIRROR` setting), indicating that under testing, `slave` should be treated as a mirror of `default`.

When the test environment is configured, a test version of `slave` will *not* be created. Instead the connection to `slave` will be redirected to point at `default`. As a result, writes to `default` will appear on `slave` – but because they are actually the same database, not because there is data replication between the two databases.

Controlling creation order for test databases

New in version 1.3: *Please see the release notes* By default, Django will always create the `default` database first. However, no guarantees are made on the creation order of any other databases in your test setup.

If your database configuration requires a specific creation order, you can specify the dependencies that exist using the `TEST_DEPENDENCIES` setting. Consider the following (simplified) example database configuration:

```
DATABASES = {
    'default': {
         # ... db settings
         'TEST_DEPENDENCIES': ['diamonds']
    },
    'diamonds': {
        # ... db settings
    },
    'clubs': {
        # ... db settings
        'TEST_DEPENDENCIES': ['diamonds']
    },
    'spades': {
        # ... db settings
        'TEST_DEPENDENCIES': ['diamonds','hearts']
    },
    'hearts': {
        # ... db settings
        'TEST_DEPENDENCIES': ['diamonds','clubs']
    }
}
```

Under this configuration, the `diamonds` database will be created first, as it is the only database alias without dependencies. The `default` and `clubs` alias will be created next (although the order of creation of this pair is not guaranteed); then `hearts`; and finally `spades`.

If there are any circular dependencies in the `TEST_DEPENDENCIES` definition, an `ImproperlyConfigured` exception will be raised.

Order in which tests are executed

In order to guarantee that all `TestCase` code starts with a clean database, the Django test runner reorders tests in the following way:

- First, all unittests (including `unittest.TestCase`, `SimpleTestCase`, `TestCase` and `TransactionTestCase`) are run with no particular ordering guaranteed nor enforced among them.
- Then any other tests (e.g. doctests) that may alter the database without restoring it to its original state are run.

Changed in version 1.5: Before Django 1.5, the only guarantee was that `TestCase` tests were always ran first, before any other tests.

Note: The new ordering of tests may reveal unexpected dependencies on test case ordering. This is the case with doctests that relied on state left in the database by a given `TransactionTestCase` test, they must be updated to be able to run independently.

Other test conditions

Regardless of the value of the `DEBUG` setting in your configuration file, all Django tests run with `DEBUG`=False. This is to ensure that the observed output of your code matches what will be seen in a production setting.

Understanding the test output

When you run your tests, you'll see a number of messages as the test runner prepares itself. You can control the level of detail of these messages with the `verbosity` option on the command line:

```
Creating test database...
Creating table myapp_animal
Creating table myapp_mineral
Loading 'initial_data' fixtures...
No fixtures found.
```

This tells you that the test runner is creating a test database, as described in the previous section.

Once the test database has been created, Django will run your tests. If everything goes well, you'll see something like this:

```
----------------------------------------------------------------------
Ran 22 tests in 0.221s

OK
```

If there are test failures, however, you'll see full details about which tests failed:

```
======================================================================
FAIL: Doctest: ellington.core.throttle.models
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/dev/django/test/doctest.py", line 2153, in runTest
    raise self.failureException(self.format_failure(new.getvalue()))
AssertionError: Failed doctest test for myapp.models
  File "/dev/myapp/models.py", line 0, in models

----------------------------------------------------------------------
File "/dev/myapp/models.py", line 14, in myapp.models
Failed example:
    throttle.check("actor A", "action one", limit=2, hours=1)
Expected:
    True
Got:
    False
```

```
----------------------------------------------------------------------
Ran 2 tests in 0.048s

FAILED (failures=1)
```

A full explanation of this error output is beyond the scope of this document, but it's pretty intuitive. You can consult the documentation of Python's `unittest` library for details.

Note that the return code for the test-runner script is 1 for any number of failed and erroneous tests. If all the tests pass, the return code is 0. This feature is useful if you're using the test-runner script in a shell script and need to test for success or failure at that level.

Speeding up the tests

In recent versions of Django, the default password hasher is rather slow by design. If during your tests you are authenticating many users, you may want to use a custom settings file and set the `PASSWORD_HASHERS` setting to a faster hashing algorithm:

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.MD5PasswordHasher',
)
```

Don't forget to also include in `PASSWORD_HASHERS` any hashing algorithm used in fixtures, if any.

3.8.3 Testing tools

Django provides a small set of tools that come in handy when writing tests.

The test client

The test client is a Python class that acts as a dummy Web browser, allowing you to test your views and interact with your Django-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate GET and POST requests on a URL and observe the response – everything from low-level HTTP (result headers and status codes) to page content.
- Test that the correct view is executed for a given URL.
- Test that a given request is rendered by a given Django template, with a template context that contains certain values.

Note that the test client is not intended to be a replacement for Selenium or other "in-browser" frameworks. Django's test client has a different focus. In short:

- Use Django's test client to establish that the correct view is being called and that the view is collecting the correct context data.
- Use in-browser frameworks like Selenium to test *rendered* HTML and the *behavior* of Web pages, namely JavaScript functionality. Django also provides special support for those frameworks; see the section on `LiveServerTestCase` for more details.

A comprehensive test suite should use a combination of both test types.

Overview and a quick example

To use the test client, instantiate `django.test.client.Client` and retrieve Web pages:

```
>>> from django.test.client import Client
>>> c = Client()
>>> response = c.post('/login/', {'username': 'john', 'password': 'smith'})
>>> response.status_code
200
>>> response = c.get('/customer/details/')
>>> response.content
'<!DOCTYPE html...'
```

As this example suggests, you can instantiate `Client` from within a session of the Python interactive interpreter.

Note a few important things about how the test client works:

- The test client does *not* require the Web server to be running. In fact, it will run just fine with no Web server running at all! That's because it avoids the overhead of HTTP and deals directly with the Django framework. This helps make the unit tests run quickly.
- When retrieving pages, remember to specify the *path* of the URL, not the whole domain. For example, this is correct:

  ```
  >>> c.get('/login/')
  ```

  This is incorrect:

  ```
  >>> c.get('http://www.example.com/login/')
  ```

  The test client is not capable of retrieving Web pages that are not powered by your Django project. If you need to retrieve other Web pages, use a Python standard library module such as `urllib` or `urllib2`.
- To resolve URLs, the test client uses whatever URLconf is pointed-to by your `ROOT_URLCONF` setting.
- Although the above example would work in the Python interactive interpreter, some of the test client's functionality, notably the template-related functionality, is only available *while tests are running*.

  The reason for this is that Django's test runner performs a bit of black magic in order to determine which template was loaded by a given view. This black magic (essentially a patching of Django's template system in memory) only happens during test running.
- By default, the test client will disable any CSRF checks performed by your site.

  If, for some reason, you *want* the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks. To do this, pass in the `enforce_csrf_checks` argument when you construct your client:

  ```
  >>> from django.test import Client
  >>> csrf_client = Client(enforce_csrf_checks=True)
  ```

Making requests

Use the `django.test.client.Client` class to make requests.

class **`Client`**(*enforce_csrf_checks=False*, ***defaults*)

It requires no arguments at time of construction. However, you can use keywords arguments to specify some default headers. For example, this will send a `User-Agent` HTTP header in each request:

```
>>> c = Client(HTTP_USER_AGENT='Mozilla/5.0')
```

The values from the `extra` keywords arguments passed to `get()`, `post()`, etc. have precedence over the defaults passed to the class constructor.

The `enforce_csrf_checks` argument can be used to test CSRF protection (see above).

Once you have a `Client` instance, you can call any of the following methods:

get(*path*, *data={}*, *follow=False*, ***extra*)

Makes a GET request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `data` dictionary are used to create a GET data payload. For example:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7})
```

...will result in the evaluation of a GET request equivalent to:

```
/customers/details/?name=fred&age=7
```

The `extra` keyword arguments parameter can be used to specify headers to be sent in the request. For example:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7},
...       HTTP_X_REQUESTED_WITH='XMLHttpRequest')
```

...will send the HTTP header `HTTP_X_REQUESTED_WITH` to the details view, which is a good way to test code paths that use the `django.http.HttpRequest.is_ajax()` method.

CGI specification

The headers sent via `**extra` should follow CGI specification. For example, emulating a different "Host" header as sent in the HTTP request from the browser to the server should be passed as `HTTP_HOST`.

If you already have the GET arguments in URL-encoded form, you can use that encoding instead of using the data argument. For example, the previous GET request could also be posed as:

```
>>> c = Client()
>>> c.get('/customers/details/?name=fred&age=7')
```

If you provide a URL with both an encoded GET data and a data argument, the data argument will take precedence.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you had an url `/redirect_me/` that redirected to `/next/`, that redirected to `/final/`, this is what you'd see:

```
>>> response = c.get('/redirect_me/', follow=True)
>>> response.redirect_chain
[(u'http://testserver/next/', 302), (u'http://testserver/final/', 302)]
```

post(*path*, *data={}*, *content_type=MULTIPART_CONTENT*, *follow=False*, ***extra*)

Makes a POST request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `data` dictionary are used to submit POST data. For example:

```
>>> c = Client()
>>> c.post('/login/', {'name': 'fred', 'passwd': 'secret'})
```

...will result in the evaluation of a POST request to this URL:

```
/login/
```

...with this POST data:

```
name=fred&passwd=secret
```

If you provide `content_type` (e.g. *text/xml* for an XML payload), the contents of `data` will be sent as-is in the POST request, using `content_type` in the HTTP `Content-Type` header.

If you don't provide a value for `content_type`, the values in `data` will be transmitted with a content type of *multipart/form-data*. In this case, the key-value pairs in `data` will be encoded as a multipart message and used to create the POST data payload.

To submit multiple values for a given key – for example, to specify the selections for a `<select multiple>` – provide the values as a list or tuple for the required key. For example, this value of `data` would submit three selected values for the field named `choices`:

```
{'choices': ('a', 'b', 'd')}
```

Submitting files is a special case. To POST a file, you need only provide the file field name as a key, and a file handle to the file you wish to upload as a value. For example:

```
>>> c = Client()
>>> with open('wishlist.doc') as fp:
...     c.post('/customers/wishes/', {'name': 'fred', 'attachment': fp})
```

(The name `attachment` here is not relevant; use whatever name your file-processing code expects.)

Note that if you wish to use the same file handle for multiple `post()` calls then you will need to manually reset the file pointer between posts. The easiest way to do this is to manually close the file after it has been provided to `post()`, as demonstrated above.

You should also ensure that the file is opened in a way that allows the data to be read. If your file contains binary data such as an image, this means you will need to open the file in `rb` (read binary) mode.

The `extra` argument acts the same as for `Client.get()`.

If the URL you request with a POST contains encoded parameters, these parameters will be made available in the request.GET data. For example, if you were to make the request:

```
>>> c.post('/login/?visitor=true', {'name': 'fred', 'passwd': 'secret'})
```

... the view handling this request could interrogate request.POST to retrieve the username and password, and could interrogate request.GET to determine if the user was a visitor.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

head(*path*, *data={}*, *follow=False*, ***extra*)
Makes a HEAD request on the provided `path` and returns a `Response` object. This method works just like `Client.get()`, including the `follow` and `extra` arguments, except it does not return a message body.

options(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, ***extra*)
Makes an OPTIONS request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`. Changed in version 1.5: `Client.options()` used to process `data` like `Client.get()`. The `follow` and `extra` arguments act the same as for `Client.get()`.

put(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, ***extra*)
Makes a PUT request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`. Changed in version 1.5: `Client.put()` used to process `data` like `Client.post()`. The `follow` and `extra` arguments act the same as for `Client.get()`.

delete(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, ***extra*)
Makes an DELETE request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`. Changed in version 1.5: `Client.delete()` used to process `data` like `Client.get()`. The `follow` and `extra` arguments act the same as for `Client.get()`.

login(***credentials*)
If your site uses Django's *authentication system* and you deal with logging in users, you can use the test client's `login()` method to simulate the effect of a user logging into the site.

After you call this method, the test client will have all the cookies and session data required to pass any login-based tests that may form part of a view.

The format of the `credentials` argument depends on which *authentication backend* you're using (which is configured by your `AUTHENTICATION_BACKENDS` setting). If you're using the standard authentication backend provided by Django (`ModelBackend`), `credentials` should be the user's username and password, provided as keyword arguments:

```
>>> c = Client()
>>> c.login(username='fred', password='secret')

# Now you can access a view that's only available to logged-in users.
```

If you're using a different authentication backend, this method may require different credentials. It requires whichever credentials are required by your backend's `authenticate()` method.

`login()` returns `True` if it the credentials were accepted and login was successful.

Finally, you'll need to remember to create user accounts before you can use this method. As we explained above, the test runner is executed using a test database, which contains no users by default. As a result, user accounts that are valid on your production site will not work under test conditions. You'll need to create users as part of the test suite – either manually (using the Django model API) or with a test fixture. Remember that if you want your test user to have a password, you can't set the user's password by setting the password attribute directly – you must use the `set_password()` function to store a correctly hashed password. Alternatively, you can use the `create_user()` helper method to create a new user with a correctly hashed password.

logout()
If your site uses Django's *authentication system*, the `logout()` method can be used to simulate the effect of a user logging out of your site.

After you call this method, the test client will have all the cookies and session data cleared to defaults. Subsequent requests will appear to come from an AnonymousUser.

Testing responses

The `get()` and `post()` methods both return a `Response` object. This `Response` object is *not* the same as the `HttpResponse` object returned Django views; the test response object has some additional data useful for test code to verify.

Specifically, a `Response` object has the following attributes:

class Response

client
The test client that was used to make the request that resulted in the response.

content
The body of the response, as a string. This is the final page content as rendered by the view, or any error message.

context
The template `Context` instance that was used to render the template that produced the response content.

If the rendered page used multiple templates, then `context` will be a list of `Context` objects, in the order in which they were rendered.

Regardless of the number of templates used during rendering, you can retrieve context values using the `[]` operator. For example, the context variable `name` could be retrieved using:

```
>>> response = client.get('/foo/')
>>> response.context['name']
'Arthur'
```

request
The request data that stimulated the response.

status_code
The HTTP status of the response, as an integer. See **RFC 2616** for a full list of HTTP status codes.

New in version 1.3: *Please see the release notes*

templates
A list of `Template` instances used to render the final content, in the order they were rendered. For each template in the list, use `template.name` to get the template's file name, if the template was loaded from a file. (The name is a string such as `'admin/index.html'`.)

You can also use dictionary syntax on the response object to query the value of any settings in the HTTP headers. For example, you could determine the content type of a response using `response['Content-Type']`.

Exceptions

If you point the test client at a view that raises an exception, that exception will be visible in the test case. You can then use a standard `try ... except` block or `assertRaises()` to test for exceptions.

The only exceptions that are not visible to the test client are `Http404`, `PermissionDenied` and `SystemExit`. Django catches these exceptions internally and converts them into the appropriate HTTP response codes. In these cases, you can check `response.status_code` in your test.

Persistent state

The test client is stateful. If a response returns a cookie, then that cookie will be stored in the test client and sent with all subsequent `get()` and `post()` requests.

Expiration policies for these cookies are not followed. If you want a cookie to expire, either delete it manually or create a new `Client` instance (which will effectively delete all cookies).

A test client has two attributes that store persistent state information. You can access these properties as part of a test condition.

`Client.`**`cookies`**
: A Python `SimpleCookie` object, containing the current values of all the client cookies. See the documentation of the `Cookie` module for more.

`Client.`**`session`**
: A dictionary-like object containing session information. See the *session documentation* for full details.

  To modify the session and then save it, it must be stored in a variable first (because a new `SessionStore` is created every time this property is accessed):

```
def test_something(self):
    session = self.client.session
    session['somekey'] = 'test'
    session.save()
```

Example

The following is a simple unit test using the test client:

```
from django.utils import unittest
from django.test.client import Client

class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Every test needs a client.
        self.client = Client()

    def test_details(self):
        # Issue a GET request.
        response = self.client.get('/customer/details/')

        # Check that the response is 200 OK.
        self.assertEqual(response.status_code, 200)

        # Check that the rendered context contains 5 customers.
        self.assertEqual(len(response.context['customers']), 5)
```

The request factory

class `RequestFactory`

New in version 1.3: *Please see the release notes* The `RequestFactory` shares the same API as the test client. However, instead of behaving like a browser, the RequestFactory provides a way to generate a request instance that can be used as the first argument to any view. This means you can test a view function the same way as you would test any other function – as a black box, with exactly known inputs, testing for specific outputs.

The API for the `RequestFactory` is a slightly restricted subset of the test client API:

- It only has access to the HTTP methods `get()`, `post()`, `put()`, `delete()`, `head()` and `options()`.
- These methods accept all the same arguments *except* for `follows`. Since this is just a factory for producing requests, it's up to you to handle the response.
- It does not support middleware. Session and authentication attributes must be supplied by the test itself if required for the view to function properly.

Example

The following is a simple unit test using the request factory:

```
from django.utils import unittest
from django.test.client import RequestFactory

class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Every test needs access to the request factory.
        self.factory = RequestFactory()

    def test_details(self):
        # Create an instance of a GET request.
        request = self.factory.get('/customer/details')

        # Test my_view() as if it were deployed at /customer/details
        response = my_view(request)
        self.assertEqual(response.status_code, 200)
```

Test cases

Provided test case classes

Normal Python unit test classes extend a base class of `unittest.TestCase`. Django provides a few extensions of this base class:

TestCase
class TestCase
This class provides some additional capabilities that can be useful for testing Web sites.

Converting a normal `unittest.TestCase` to a Django `TestCase` is easy: Just change the base class of your test from *'unittest.TestCase'* to *'django.test.TestCase'*. All of the standard Python unit test functionality will continue to be available, but it will be augmented with some useful additions, including:

- Automatic loading of fixtures.
- Wraps each test in a transaction.
- Creates a TestClient instance.
- Django-specific assertions for testing for things like redirection and form errors.

Changed in version 1.5: The order in which tests are run has changed. See Order in which tests are executed. `TestCase` inherits from `TransactionTestCase`.

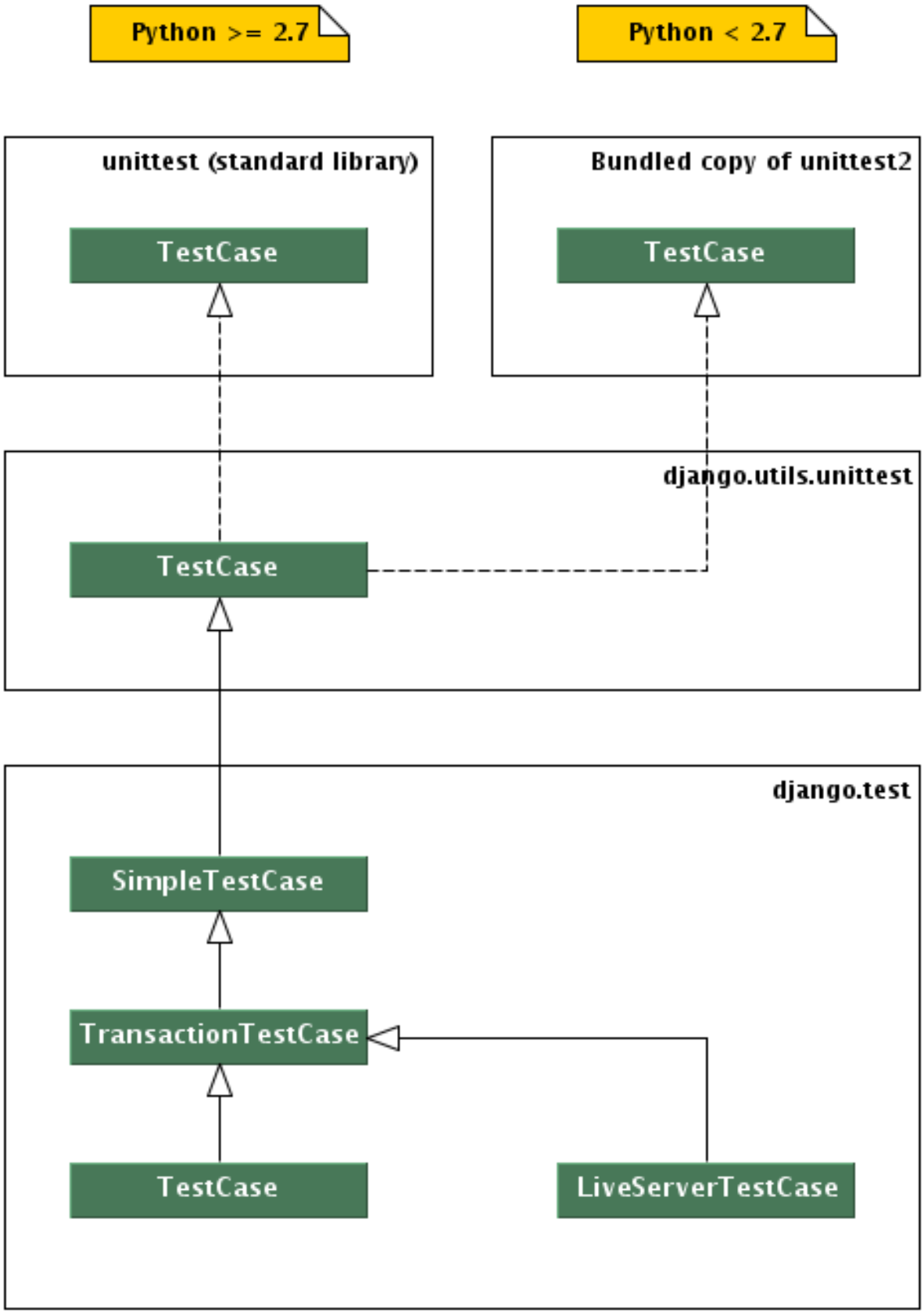


Figure 3.1: Hierarchy of Django unit testing classes

TransactionTestCase

class TransactionTestCase

Django `TestCase` classes make use of database transaction facilities, if available, to speed up the process of resetting the database to a known state at the beginning of each test. A consequence of this, however, is that the effects of transaction commit and rollback cannot be tested by a Django `TestCase` class. If your test requires testing of such transactional behavior, you should use a Django `TransactionTestCase`.

`TransactionTestCase` and `TestCase` are identical except for the manner in which the database is reset to a known state and the ability for test code to test the effects of commit and rollback:

- A `TransactionTestCase` resets the database after the test runs by truncating all tables. A `TransactionTestCase` may call commit and rollback and observe the effects of these calls on the database.
- A `TestCase`, on the other hand, does not truncate tables after a test. Instead, it encloses the test code in a database transaction that is rolled back at the end of the test. It also prevents the code under test from issuing any commit or rollback operations on the database, to ensure that the rollback at the end of the test restores the database to its initial state.

  When running on a database that does not support rollback (e.g. MySQL with the MyISAM storage engine), `TestCase` falls back to initializing the database by truncating tables and reloading initial data.

Note: Changed in version 1.5: *Please see the release notes* Prior to 1.5, `TransactionTestCase` flushed the database tables *before* each test. In Django 1.5, this is instead done *after* the test has been run.

When the flush took place before the test, it was guaranteed that primary key values started at one in `TransactionTestCase` tests.

Tests should not depend on this behaviour, but for legacy tests that do, the `reset_sequences` attribute can be used until the test has been properly updated.

Changed in version 1.5: The order in which tests are run has changed. See Order in which tests are executed. `TransactionTestCase` inherits from `SimpleTestCase`.

`TransactionTestCase.`**reset_sequences**

New in version 1.5: *Please see the release notes* Setting `reset_sequences = True` on a `TransactionTestCase` will make sure sequences are always reset before the test run:

```
class TestsThatDependsOnPrimaryKeySequences(TransactionTestCase):
    reset_sequences = True

    def test_animal_pk(self):
        lion = Animal.objects.create(name="lion", sound="roar")
        # lion.pk is guaranteed to always be 1
        self.assertEqual(lion.pk, 1)
```

Unless you are explicitly testing primary keys sequence numbers, it is recommended that you do not hard code primary key values in tests.

Using `reset_sequences = True` will slow down the test, since the primary key reset is an relatively expensive database operation.

SimpleTestCase

class SimpleTestCase

New in version 1.4: *Please see the release notes* A very thin subclass of `unittest.TestCase`, it extends it with some basic functionality like:

- Saving and restoring the Python warning machinery state.
- Checking that a callable `raises a certain exception`.

- `Testing form field rendering`.
- Testing server *HTML responses for the presence/lack of a given fragment*.
- The ability to run tests with *modified settings*

If you need any of the other more complex and heavyweight Django-specific features like:

- Using the `client` `Client`.
- Testing or using the ORM.
- Database `fixtures`.
- Custom test-time `URL maps`.
- Test *skipping based on database backend features*.
- The remaining specialized *assert** methods.

then you should use `TransactionTestCase` or `TestCase` instead.

`SimpleTestCase` inherits from `django.utils.unittest.TestCase`.

Default test client

`TestCase.`**`client`**

Every test case in a `django.test.TestCase` instance has access to an instance of a Django test client. This client can be accessed as `self.client`. This client is recreated for each test, so you don't have to worry about state (such as cookies) carrying over from one test to another.

This means, instead of instantiating a `Client` in each test:

```
from django.utils import unittest
from django.test.client import Client

class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get('/customer/details/')
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

...you can just refer to `self.client`, like so:

```
from django.test import TestCase

class SimpleTest(TestCase):
    def test_details(self):
        response = self.client.get('/customer/details/')
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        response = self.client.get('/customer/index/')
        self.assertEqual(response.status_code, 200)
```

Customizing the test client

New in version 1.3: *Please see the release notes*

`TestCase.client_class`

If you want to use a different `Client` class (for example, a subclass with customized behavior), use the `client_class` class attribute:

```
from django.test import TestCase
from django.test.client import Client

class MyTestClient(Client):
    # Specialized methods for your environment...

class MyTest(TestCase):
    client_class = MyTestClient

    def test_my_stuff(self):
        # Here self.client is an instance of MyTestClient...
```

Fixture loading

`TestCase.fixtures`

A test case for a database-backed Web site isn't much use if there isn't any data in the database. To make it easy to put test data into the database, Django's custom `TestCase` class provides a way of loading **fixtures**.

A fixture is a collection of data that Django knows how to import into a database. For example, if your site has user accounts, you might set up a fixture of fake user accounts in order to populate your database during tests.

The most straightforward way of creating a fixture is to use the `manage.py dumpdata` command. This assumes you already have some data in your database. See the `dumpdata documentation` for more details.

Note: If you've ever run `manage.py syncdb`, you've already used a fixture without even knowing it! When you call `syncdb` in the database for the first time, Django installs a fixture called `initial_data`. This gives you a way of populating a new database with any initial data, such as a default set of categories.

Fixtures with other names can always be installed manually using the `manage.py loaddata` command.

Initial SQL data and testing

Django provides a second way to insert initial data into models – the *custom SQL hook*. However, this technique *cannot* be used to provide initial data for testing purposes. Django's test framework flushes the contents of the test database after each test; as a result, any data added using the custom SQL hook will be lost.

Once you've created a fixture and placed it in a `fixtures` directory in one of your `INSTALLED_APPS`, you can use it in your unit tests by specifying a `fixtures` class attribute on your `django.test.TestCase` subclass:

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    fixtures = ['mammals.json', 'birds']

    def setUp(self):
```

```
        # Test definitions as before.
        call_setup_methods()

    def testFluffyAnimals(self):
        # A test that uses the fixtures.
        call_some_test_code()
```

Here's specifically what will happen:

- At the start of each test case, before `setUp()` is run, Django will flush the database, returning the database to the state it was in directly after `syncdb` was called.
- Then, all the named fixtures are installed. In this example, Django will install any JSON fixture named `mammals`, followed by any fixture named `birds`. See the `loaddata` documentation for more details on defining and installing fixtures.

This flush/load procedure is repeated for each test in the test case, so you can be certain that the outcome of a test will not be affected by another test, or by the order of test execution.

URLconf configuration

`TestCase.`**`urls`**

If your application provides views, you may want to include tests that use the test client to exercise those views. However, an end user is free to deploy the views in your application at any URL of their choosing. This means that your tests can't rely upon the fact that your views will be available at a particular URL.

In order to provide a reliable URL space for your test, `django.test.TestCase` provides the ability to customize the URLconf configuration for the duration of the execution of a test suite. If your `TestCase` instance defines an `urls` attribute, the `TestCase` will use the value of that attribute as the `ROOT_URLCONF` for the duration of that test.

For example:

```
from django.test import TestCase

class TestMyViews(TestCase):
    urls = 'myapp.test_urls'

    def testIndexPageView(self):
        # Here you'd test your view using ''Client''.
        call_some_test_code()
```

This test case will use the contents of `myapp.test_urls` as the URLconf for the duration of the test case.

Multi-database support

`TestCase.`**`multi_db`**

Django sets up a test database corresponding to every database that is defined in the `DATABASES` definition in your settings file. However, a big part of the time taken to run a Django TestCase is consumed by the call to `flush` that ensures that you have a clean database at the start of each test run. If you have multiple databases, multiple flushes are required (one for each database), which can be a time consuming activity – especially if your tests don't need to test multi-database activity.

As an optimization, Django only flushes the `default` database at the start of each test run. If your setup contains multiple databases, and you have a test that requires every database to be clean, you can use the `multi_db` attribute on the test suite to request a full flush.

For example:

```
class TestMyViews(TestCase):
    multi_db = True

    def testIndexPageView(self):
        call_some_test_code()
```

This test case will flush *all* the test databases before running `testIndexPageView`.

Overriding settings

`TestCase.settings()`

New in version 1.4: *Please see the release notes* For testing purposes it's often useful to change a setting temporarily and revert to the original value after running the testing code. For this use case Django provides a standard Python context manager (see PEP 343) `settings()`, which can be used like this:

```
from django.test import TestCase

class LoginTestCase(TestCase):

    def test_login(self):

        # First check for the default behavior
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/accounts/login/?next=/sekrit/')

        # Then override the LOGIN_URL setting
        with self.settings(LOGIN_URL='/other/login/'):
            response = self.client.get('/sekrit/')
            self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

This example will override the `LOGIN_URL` setting for the code in the `with` block and reset its value to the previous state afterwards.

override_settings()

In case you want to override a setting for just one test method or even the whole `TestCase` class, Django provides the `override_settings()` decorator (see PEP 318). It's used like this:

```
from django.test import TestCase
from django.test.utils import override_settings

class LoginTestCase(TestCase):

    @override_settings(LOGIN_URL='/other/login/')
    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

The decorator can also be applied to test case classes:

```
from django.test import TestCase
from django.test.utils import override_settings

@override_settings(LOGIN_URL='/other/login/')
class LoginTestCase(TestCase):
```

```
    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

Note: When given a class, the decorator modifies the class directly and returns it; it doesn't create and return a modified copy of it. So if you try to tweak the above example to assign the return value to a different name than `LoginTestCase`, you may be surprised to find that the original `LoginTestCase` is still equally affected by the decorator.

Note: When overriding settings, make sure to handle the cases in which your app's code uses a cache or similar feature that retains state even if the setting is changed. Django provides the `django.test.signals.setting_changed` signal that lets you register callbacks to clean up and otherwise reset state when settings are changed. Note that this signal isn't currently used by Django itself, so changing built-in settings may not yield the results you expect.

Emptying the test outbox

If you use Django's custom `TestCase` class, the test runner will clear the contents of the test email outbox at the start of each test case.

For more detail on email services during tests, see Email services.

Assertions

As Python's normal `unittest.TestCase` class implements assertion methods such as `assertTrue()` and `assertEqual()`, Django's custom `TestCase` class provides a number of custom assertion methods that are useful for testing Web applications:

The failure messages given by most of these assertion methods can be customized with the `msg_prefix` argument. This string will be prefixed to any failure message generated by the assertion. This allows you to provide additional details that may help you to identify the location and cause of an failure in your test suite.

`SimpleTestCase.`**`assertRaisesMessage`**(*expected_exception*, *expected_message*, *callable_obj=None*, **args*, ***kwargs*)

New in version 1.4: *Please see the release notes* Asserts that execution of callable `callable_obj` raised the `expected_exception` exception and that such exception has an `expected_message` representation. Any other outcome is reported as a failure. Similar to unittest's `assertRaisesRegexp()` with the difference that `expected_message` isn't a regular expression.

`SimpleTestCase.`**`assertFieldOutput`**(*self*, *fieldclass*, *valid*, *invalid*, *field_args=None*, *field_kwargs=None*, *empty_value=u''*)

New in version 1.4: *Please see the release notes* Asserts that a form field behaves correctly with various inputs.

Parameters

- **fieldclass** – the class of the field to be tested.
- **valid** – a dictionary mapping valid inputs to their expected cleaned values.
- **invalid** – a dictionary mapping invalid inputs to one or more raised error messages.
- **field_args** – the args passed to instantiate the field.
- **field_kwargs** – the kwargs passed to instantiate the field.

- **empty_value** – the expected clean output for inputs in `EMPTY_VALUES`.

For example, the following code tests that an `EmailField` accepts "a@a.com" as a valid email address, but rejects "aaa" with a reasonable error message:

```
self.assertFieldOutput(EmailField, {'a@a.com': 'a@a.com'}, {'aaa': [u'Enter a valid e-mail addre
```

`TestCase.`**`assertContains`**(*response*, *text*, *count=None*, *status_code=200*, *msg_prefix=''*, *html=False*)

Asserts that a `Response` instance produced the given `status_code` and that `text` appears in the content of the response. If `count` is provided, `text` must occur exactly `count` times in the response. New in version 1.4: *Please see the release notes* Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See `assertHTMLEqual()` for more details.

`TestCase.`**`assertNotContains`**(*response*, *text*, *status_code=200*, *msg_prefix=''*, *html=False*)

Asserts that a `Response` instance produced the given `status_code` and that `text` does not appears in the content of the response. New in version 1.4: *Please see the release notes* Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See `assertHTMLEqual()` for more details.

`TestCase.`**`assertFormError`**(*response*, *form*, *field*, *errors*, *msg_prefix=''*)

Asserts that a field on a form raises the provided list of errors when rendered on the form.

`form` is the name the `Form` instance was given in the template context.

`field` is the name of the field on the form to check. If `field` has a value of `None`, non-field errors (errors you can access via `form.non_field_errors()`) will be checked.

`errors` is an error string, or a list of error strings, that are expected as a result of form validation.

`TestCase.`**`assertTemplateUsed`**(*response*, *template_name*, *msg_prefix=''*)

Asserts that the template with the given name was used in rendering the response.

The name is a string such as `'admin/index.html'`. New in version 1.4: *Please see the release notes* You can use this as a context manager, like this:

```
with self.assertTemplateUsed('index.html'):
    render_to_string('index.html')
with self.assertTemplateUsed(template_name='index.html'):
    render_to_string('index.html')
```

`TestCase.`**`assertTemplateNotUsed`**(*response*, *template_name*, *msg_prefix=''*)

Asserts that the template with the given name was *not* used in rendering the response. New in version 1.4: *Please see the release notes* You can use this as a context manager in the same way as `assertTemplateUsed()`.

`TestCase.`**`assertRedirects`**(*response*, *expected_url*, *status_code=302*, *target_status_code=200*, *msg_prefix=''*)

Asserts that the response return a `status_code` redirect status, it redirected to `expected_url` (including any GET data), and the final page was received with `target_status_code`.

If your request used the `follow` argument, the `expected_url` and `target_status_code` will be the url and status code for the final point of the redirect chain.

`TestCase.`**`assertQuerysetEqual`**(*qs*, *values*, *transform=repr*, *ordered=True*)

New in version 1.3: *Please see the release notes* Asserts that a queryset `qs` returns a particular list of values `values`.

The comparison of the contents of `qs` and `values` is performed using the function `transform`; by default, this means that the `repr()` of each value is compared. Any other callable can be used if `repr()` doesn't provide a unique or helpful comparison.

By default, the comparison is also ordering dependent. If `qs` doesn't provide an implicit ordering, you can set the `ordered` parameter to `False`, which turns the comparison into a Python set comparison. Changed in version 1.4: The `ordered` parameter is new in version 1.4. In earlier versions, you would need to ensure the queryset is ordered consistently, possibly via an explicit `order_by()` call on the queryset prior to comparison.

`TestCase.`**`assertNumQueries`**(*num*, *func*, **args*, ***kwargs*)

New in version 1.3: *Please see the release notes* Asserts that when `func` is called with `*args` and `**kwargs` that `num` database queries are executed.

If a `"using"` key is present in `kwargs` it is used as the database alias for which to check the number of queries. If you wish to call a function with a `using` parameter you can do it by wrapping the call with a `lambda` to add an extra parameter:

```
self.assertNumQueries(7, lambda: my_function(using=7))
```

You can also use this as a context manager:

```
with self.assertNumQueries(2):
    Person.objects.create(name="Aaron")
    Person.objects.create(name="Daniel")
```

`SimpleTestCase.`**`assertHTMLEqual`**(*html1*, *html2*, *msg=None*)

New in version 1.4: *Please see the release notes* Asserts that the strings `html1` and `html2` are equal. The comparison is based on HTML semantics. The comparison takes following things into account:

- Whitespace before and after HTML tags is ignored.
- All types of whitespace are considered equivalent.
- All open tags are closed implicitly, e.g. when a surrounding tag is closed or the HTML document ends.
- Empty tags are equivalent to their self-closing version.
- The ordering of attributes of an HTML element is not significant.
- Attributes without an argument are equal to attributes that equal in name and value (see the examples).

The following examples are valid tests and don't raise any `AssertionError`:

```
self.assertHTMLEqual('<p>Hello <b>world!</p>',
    '''<p>
        Hello   <b>world! <b/>
    </p>''')
self.assertHTMLEqual(
    '<input type="checkbox" checked="checked" id="id_accept_terms" />',
    '<input id="id_accept_terms" type='checkbox' checked>')
```

`html1` and `html2` must be valid HTML. An `AssertionError` will be raised if one of them cannot be parsed.

`SimpleTestCase.`**`assertHTMLNotEqual`**(*html1*, *html2*, *msg=None*)

New in version 1.4: *Please see the release notes* Asserts that the strings `html1` and `html2` are *not* equal. The comparison is based on HTML semantics. See `assertHTMLEqual()` for details.

`html1` and `html2` must be valid HTML. An `AssertionError` will be raised if one of them cannot be parsed.

Email services

If any of your Django views send email using *Django's email functionality*, you probably don't want to send email each time you run a test using that view. For this reason, Django's test runner automatically redirects all Django-sent

email to a dummy outbox. This lets you test every aspect of sending email – from the number of messages sent to the contents of each message – without actually sending the messages.

The test runner accomplishes this by transparently replacing the normal email backend with a testing backend. (Don't worry – this has no effect on any other email senders outside of Django, such as your machine's mail server, if you're running one.)

`django.core.mail.outbox`

During test running, each outgoing email is saved in `django.core.mail.outbox`. This is a simple list of all `EmailMessage` instances that have been sent. The `outbox` attribute is a special attribute that is created *only* when the `locmem` email backend is used. It doesn't normally exist as part of the `django.core.mail` module and you can't import it directly. The code below shows how to access this attribute correctly.

Here's an example test that examines `django.core.mail.outbox` for length and contents:

```
from django.core import mail
from django.test import TestCase

class EmailTest(TestCase):
    def test_send_email(self):
        # Send message.
        mail.send_mail('Subject here', 'Here is the message.',
            'from@example.com', ['to@example.com'],
            fail_silently=False)

        # Test that one message has been sent.
        self.assertEqual(len(mail.outbox), 1)

        # Verify that the subject of the first message is correct.
        self.assertEqual(mail.outbox[0].subject, 'Subject here')
```

As noted *previously*, the test outbox is emptied at the start of every test in a Django `TestCase`. To empty the outbox manually, assign the empty list to `mail.outbox`:

```
from django.core import mail

# Empty the test outbox
mail.outbox = []
```

Skipping tests

New in version 1.3: *Please see the release notes* The unittest library provides the `@skipIf` and `@skipUnless` decorators to allow you to skip tests if you know ahead of time that those tests are going to fail under certain conditions.

For example, if your test requires a particular optional library in order to succeed, you could decorate the test case with `@skipIf`. Then, the test runner will report that the test wasn't executed and why, instead of failing the test or omitting the test altogether.

To supplement these test skipping behaviors, Django provides two additional skip decorators. Instead of testing a generic boolean, these decorators check the capabilities of the database, and skip the test if the database doesn't support a specific named feature.

The decorators use a string identifier to describe database features. This string corresponds to attributes of the database connection features class. See `BaseDatabaseFeatures` class for a full list of database features that can be used as a basis for skipping tests.

skipIfDBFeature(*feature_name_string*)

Skip the decorated test if the named database feature is supported.

For example, the following test will not be executed if the database supports transactions (e.g., it would *not* run under PostgreSQL, but it would under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipIfDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
```

skipUnlessDBFeature(*feature_name_string*)

Skip the decorated test if the named database feature is *not* supported.

For example, the following test will only be executed if the database supports transactions (e.g., it would run under PostgreSQL, but *not* under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipUnlessDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
```

Live test server

New in version 1.4: *Please see the release notes*

class LiveServerTestCase

`LiveServerTestCase` does basically the same as `TransactionTestCase` with one extra feature: it launches a live Django server in the background on setup, and shuts it down on teardown. This allows the use of automated test clients other than the *Django dummy client* such as, for example, the Selenium client, to execute a series of functional tests inside a browser and simulate a real user's actions.

By default the live server's address is *'localhost:8081'* and the full URL can be accessed during the tests with `self.live_server_url`. If you'd like to change the default address (in the case, for example, where the 8081 port is already taken) then you may pass a different one to the `test` command via the `--liveserver` option, for example:

```
./manage.py test --liveserver=localhost:8082
```

Another way of changing the default server address is by setting the *DJANGO_LIVE_TEST_SERVER_ADDRESS* environment variable somewhere in your code (for example, in a *custom test runner*):

```
import os
os.environ['DJANGO_LIVE_TEST_SERVER_ADDRESS'] = 'localhost:8082'
```

In the case where the tests are run by multiple processes in parallel (for example, in the context of several simultaneous continuous integration builds), the processes will compete for the same address, and therefore your tests might randomly fail with an "Address already in use" error. To avoid this problem, you can pass a comma-separated list of ports or ranges of ports (at least as many as the number of potential parallel processes). For example:

```
./manage.py test --liveserver=localhost:8082,8090-8100,9000-9200,7041
```

Then, during test execution, each new live test server will try every specified port until it finds one that is free and takes it.

To demonstrate how to use `LiveServerTestCase`, let's write a simple Selenium test. First of all, you need to install the selenium package into your Python path:

```
pip install selenium
```

Then, add a `LiveServerTestCase`-based test to your app's tests module (for example: `myapp/tests.py`). The code for this test may look as follows:

```
from django.test import LiveServerTestCase
from selenium.webdriver.firefox.webdriver import WebDriver

class MySeleniumTests(LiveServerTestCase):
    fixtures = ['user-data.json']

    @classmethod
    def setUpClass(cls):
        cls.selenium = WebDriver()
        super(MySeleniumTests, cls).setUpClass()

    @classmethod
    def tearDownClass(cls):
        super(MySeleniumTests, cls).tearDownClass()
        cls.selenium.quit()

    def test_login(self):
        self.selenium.get('%s%s' % (self.live_server_url, '/login/'))
        username_input = self.selenium.find_element_by_name("username")
        username_input.send_keys('myuser')
        password_input = self.selenium.find_element_by_name("password")
        password_input.send_keys('secret')
        self.selenium.find_element_by_xpath('//input[@value="Log in"]').click()
```

Finally, you may run the test as follows:

```
./manage.py test myapp.MySeleniumTests.test_login
```

This example will automatically open Firefox then go to the login page, enter the credentials and press the "Log in" button. Selenium offers other drivers in case you do not have Firefox installed or wish to use another browser. The example above is just a tiny fraction of what the Selenium client can do; check out the full reference for more details.

Note: `LiveServerTestCase` makes use of the *staticfiles contrib app* so you'll need to have your project configured accordingly (in particular by setting `STATIC_URL`).

Note: When using an in-memory SQLite database to run the tests, the same database connection will be shared by two threads in parallel: the thread in which the live server is run and the thread in which the test case is run. It's important to prevent simultaneous database queries via this shared connection by the two threads, as that may sometimes randomly cause the tests to fail. So you need to ensure that the two threads don't access the database at the same time. In particular, this means that in some cases (for example, just after clicking a link or submitting a form), you might need to check that a response is received by Selenium and that the next page is loaded before proceeding with further test execution. Do this, for example, by making Selenium wait until the <*body*> HTML tag is found in the response (requires Selenium > 2.13):

```
def test_login(self):
    from selenium.webdriver.support.wait import WebDriverWait
    ...
    self.selenium.find_element_by_xpath('//input[@value="Log in"]').click()
    # Wait until the response is received
    WebDriverWait(self.selenium, timeout).until(
        lambda driver: driver.find_element_by_tag_name('body'), timeout=10)
```

The tricky thing here is that there's really no such thing as a "page load," especially in modern Web apps that generate HTML dynamically after the server generates the initial document. So, simply checking for the presence of *<body>* in the response might not necessarily be appropriate for all use cases. Please refer to the Selenium FAQ and Selenium documentation for more information.

3.8.4 Using different testing frameworks

Clearly, `doctest` and `unittest` are not the only Python testing frameworks. While Django doesn't provide explicit support for alternative frameworks, it does provide a way to invoke tests constructed for an alternative framework as if they were normal Django tests.

When you run `./manage.py test`, Django looks at the `TEST_RUNNER` setting to determine what to do. By default, `TEST_RUNNER` points to `'django.test.simple.DjangoTestSuiteRunner'`. This class defines the default Django testing behavior. This behavior involves:

1. Performing global pre-test setup.
2. Looking for unit tests and doctests in the `models.py` and `tests.py` files in each installed application.
3. Creating the test databases.
4. Running `syncdb` to install models and initial data into the test databases.
5. Running the unit tests and doctests that are found.
6. Destroying the test databases.
7. Performing global post-test teardown.

If you define your own test runner class and point `TEST_RUNNER` at that class, Django will execute your test runner whenever you run `./manage.py test`. In this way, it is possible to use any test framework that can be executed from Python code, or to modify the Django test execution process to satisfy whatever testing requirements you may have.

Defining a test runner

A test runner is a class defining a `run_tests()` method. Django ships with a `DjangoTestSuiteRunner` class that defines the default Django testing behavior. This class defines the `run_tests()` entry point, plus a selection of other methods that are used to by `run_tests()` to set up, execute and tear down the test suite.

class `DjangoTestSuiteRunner`(*verbosity=1*, *interactive=True*, *failfast=True*, ***kwargs*)

`verbosity` determines the amount of notification and debug information that will be printed to the console; `0` is no output, `1` is normal output, and `2` is verbose output.

If `interactive` is `True`, the test suite has permission to ask the user for instructions when the test suite is executed. An example of this behavior would be asking for permission to delete an existing test database. If `interactive` is `False`, the test suite must be able to run without any manual intervention.

If `failfast` is `True`, the test suite will stop running after the first test failure is detected.

Django will, from time to time, extend the capabilities of the test runner by adding new arguments. The `**kwargs` declaration allows for this expansion. If you subclass `DjangoTestSuiteRunner` or write your own test runner, ensure accept and handle the `**kwargs` parameter. New in version 1.4: *Please see the release notes* Your test runner may also define additional command-line options. If you add an `option_list` attribute to a subclassed test runner, those options will be added to the list of command-line options that the `test` command can use.

Attributes

DjangoTestSuiteRunner.**option_list**
New in version 1.4: *Please see the release notes* This is the tuple of `optparse` options which will be fed into the management command's `OptionParser` for parsing arguments. See the documentation for Python's `optparse` module for more details.

Methods

DjangoTestSuiteRunner.**run_tests**(*test_labels*, *extra_tests=None*, ***kwargs*)
Run the test suite.

`test_labels` is a list of strings describing the tests to be run. A test label can take one of three forms:

- `app.TestCase.test_method` – Run a single test method in a test case.
- `app.TestCase` – Run all the test methods in a test case.
- `app` – Search for and run all tests in the named application.

If `test_labels` has a value of `None`, the test runner should run search for tests in all the applications in `INSTALLED_APPS`.

`extra_tests` is a list of extra `TestCase` instances to add to the suite that is executed by the test runner. These extra tests are run in addition to those discovered in the modules listed in `test_labels`.

This method should return the number of tests that failed.

DjangoTestSuiteRunner.**setup_test_environment**(***kwargs*)
Sets up the test environment ready for testing.

DjangoTestSuiteRunner.**build_suite**(*test_labels*, *extra_tests=None*, ***kwargs*)
Constructs a test suite that matches the test labels provided.

`test_labels` is a list of strings describing the tests to be run. A test label can take one of three forms:

- `app.TestCase.test_method` – Run a single test method in a test case.
- `app.TestCase` – Run all the test methods in a test case.
- `app` – Search for and run all tests in the named application.

If `test_labels` has a value of `None`, the test runner should run search for tests in all the applications in `INSTALLED_APPS`.

`extra_tests` is a list of extra `TestCase` instances to add to the suite that is executed by the test runner. These extra tests are run in addition to those discovered in the modules listed in `test_labels`.

Returns a `TestSuite` instance ready to be run.

DjangoTestSuiteRunner.**setup_databases**(***kwargs*)
Creates the test databases.

Returns a data structure that provides enough detail to undo the changes that have been made. This data will be provided to the `teardown_databases()` function at the conclusion of testing.

DjangoTestSuiteRunner.**run_suite**(*suite*, ***kwargs*)
Runs the test suite.

Returns the result produced by the running the test suite.

`DjangoTestSuiteRunner.`**`teardown_databases`**`(`*old_config*, ***kwargs*`)`
Destroys the test databases, restoring pre-test conditions.

`old_config` is a data structure defining the changes in the database configuration that need to be reversed. It is the return value of the `setup_databases()` method.

`DjangoTestSuiteRunner.`**`teardown_test_environment`**`(`***kwargs*`)`
Restores the pre-test environment.

`DjangoTestSuiteRunner.`**`suite_result`**`(`*suite*, *result*, ***kwargs*`)`
Computes and returns a return code based on a test suite, and the result from that test suite.

Testing utilities

To assist in the creation of your own test runner, Django provides a number of utility methods in the `django.test.utils` module.

`setup_test_environment``()`
Performs any global pre-test setup, such as the installing the instrumentation of the template rendering system and setting up the dummy email outbox.

`teardown_test_environment``()`
Performs any global post-test teardown, such as removing the black magic hooks into the template system and restoring normal email services.

The creation module of the database backend (`connection.creation`) also provides some utilities that can be useful during testing.

`create_test_db``(`[*verbosity=1*, *autoclobber=False*]`)`
Creates a new test database and runs `syncdb` against it.

`verbosity` has the same behavior as in `run_tests()`.

`autoclobber` describes the behavior that will occur if a database with the same name as the test database is discovered:

- If `autoclobber` is `False`, the user will be asked to approve destroying the existing database. `sys.exit` is called if the user does not approve.
- If autoclobber is `True`, the database will be destroyed without consulting the user.

Returns the name of the test database that it created.

`create_test_db()` has the side effect of modifying the value of `NAME` in `DATABASES` to match the name of the test database.

`destroy_test_db``(`*old_database_name*[, *verbosity=1*]`)`
Destroys the database whose name is the value of `NAME` in `DATABASES`, and sets `NAME` to the value of `old_database_name`.

The `verbosity` argument has the same behavior as for `DjangoTestSuiteRunner`.

3.9 User authentication in Django

Django comes with a user authentication system. It handles user accounts, groups, permissions and cookie-based user sessions. This document explains how things work.

3.9.1 Overview

The auth system consists of:

- Users
- Permissions: Binary (yes/no) flags designating whether a user may perform a certain task.
- Groups: A generic way of applying labels and permissions to more than one user.

3.9.2 Installation

Authentication support is bundled as a Django application in `django.contrib.auth`. To install it, do the following:

1. Put `'django.contrib.auth'` and `'django.contrib.contenttypes'` in your `INSTALLED_APPS` setting. (The `Permission` model in `django.contrib.auth` depends on `django.contrib.contenttypes`.)
2. Run the command `manage.py syncdb`.

Note that the default `settings.py` file created by `django-admin.py startproject` includes `'django.contrib.auth'` and `'django.contrib.contenttypes'` in `INSTALLED_APPS` for convenience. If your `INSTALLED_APPS` already contains these apps, feel free to run `manage.py syncdb` again; you can run that command as many times as you'd like, and each time it'll only install what's needed.

The `syncdb` command creates the necessary database tables, creates permission objects for all installed apps that need 'em, and prompts you to create a superuser account the first time you run it.

Once you've taken those steps, that's it.

3.9.3 Users

class `models.`**`User`**

API reference

Fields

class `models.`**`User`**

`User` objects have the following fields:

`username`
Required. 30 characters or fewer. Usernames may contain alphanumeric, `_`, `@`, `+`, `.` and `-` characters.

`first_name`
Optional. 30 characters or fewer.

`last_name`
Optional. 30 characters or fewer.

`email`
Optional. Email address.

`password`
Required. A hash of, and metadata about, the password. (Django doesn't store the raw password.) Raw passwords can be arbitrarily long and can contain any character. See the "Passwords" section below.

is_staff
Boolean. Designates whether this user can access the admin site.

is_active
Boolean. Designates whether this user account should be considered active. We recommend that you set this flag to `False` instead of deleting accounts; that way, if your applications have any foreign keys to users, the foreign keys won't break.

This doesn't necessarily control whether or not the user can log in. Authentication backends aren't required to check for the `is_active` flag, so if you want to reject a login based on `is_active` being `False`, it's up to you to check that in your own login view. However, the `AuthenticationForm` used by the `login()` view *does* perform this check, as do the permission-checking methods such as `has_perm()` and the authentication in the Django admin. All of those functions/methods will return `False` for inactive users.

is_superuser
Boolean. Designates that this user has all permissions without explicitly assigning them.

last_login
A datetime of the user's last login. Is set to the current date/time by default.

date_joined
A datetime designating when the account was created. Is set to the current date/time by default when the account is created.

Methods

class `models.`**`User`**
`User` objects have two many-to-many fields: `groups` and `user_permissions`. `User` objects can access their related objects in the same way as any other *Django model*:

```
myuser.groups = [group_list]
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions = [permission_list]
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

In addition to those automatic API methods, `User` objects have the following custom methods:

is_anonymous()
Always returns `False`. This is a way of differentiating `User` and `AnonymousUser` objects. Generally, you should prefer using `is_authenticated()` to this method.

is_authenticated()
Always returns `True`. This is a way to tell if the user has been authenticated. This does not imply any permissions, and doesn't check if the user is active - it only indicates that the user has provided a valid username and password.

get_full_name()
Returns the `first_name` plus the `last_name`, with a space in between.

set_password(*raw_password*)
Sets the user's password to the given raw string, taking care of the password hashing. Doesn't save the `User` object.

check_password(*raw_password*)
Returns `True` if the given raw string is the correct password for the user. (This takes care of the password hashing in making the comparison.)

set_unusable_password()
Marks the user as having no password set. This isn't the same as having a blank string for a password. `check_password()` for this user will never return `True`. Doesn't save the `User` object.

You may need this if authentication for your application takes place against an existing external source such as an LDAP directory.

has_usable_password()
Returns `False` if `set_unusable_password()` has been called for this user.

get_group_permissions(*obj=None*)
Returns a set of permission strings that the user has, through his/her groups.

If `obj` is passed in, only returns the group permissions for this specific object.

get_all_permissions(*obj=None*)
Returns a set of permission strings that the user has, both through group and user permissions.

If `obj` is passed in, only returns the permissions for this specific object.

has_perm(*perm*, *obj=None*)
Returns `True` if the user has the specified permission, where perm is in the format `"<app label>.<permission codename>"`. (see permissions section below). If the user is inactive, this method will always return `False`.

If `obj` is passed in, this method won't check for a permission for the model, but for this specific object.

has_perms(*perm_list*, *obj=None*)
Returns `True` if the user has each of the specified permissions, where each perm is in the format `"<app label>.<permission codename>"`. If the user is inactive, this method will always return `False`.

If `obj` is passed in, this method won't check for permissions for the model, but for the specific object.

has_module_perms(*package_name*)
Returns `True` if the user has any permissions in the given package (the Django app label). If the user is inactive, this method will always return `False`.

email_user(*subject*, *message*, *from_email=None*)
Sends an email to the user. If `from_email` is `None`, Django uses the `DEFAULT_FROM_EMAIL`.

get_profile()
Returns a site-specific profile for this user. Raises `django.contrib.auth.models.SiteProfileNotAvailable` if the current site doesn't allow profiles, or `django.core.exceptions.ObjectDoesNotExist` if the user does not have a profile. For information on how to define a site-specific user profile, see the section on storing additional user information below.

Manager functions

class `models.`**UserManager**
The `User` model has a custom manager that has the following helper functions:

create_user(*username*, *email=None*, *password=None*)
Changed in version 1.4: The `email` parameter was made optional. The username parameter is now checked for emptiness and raises a `ValueError` in case of a negative result. Creates, saves and returns a `User`.

The `username` and `password` are set as given. The domain portion of `email` is automatically converted to lowercase, and the returned `User` object will have `is_active` set to `True`.

If no password is provided, `set_unusable_password()` will be called.

See Creating users for example usage.

make_random_password(*length=10*, *allowed_chars='abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ234*

Returns a random password with the given length and given string of allowed characters. (Note that the default value of `allowed_chars` doesn't contain letters that can cause user confusion, including:

- `i`, `l`, `I`, and `1` (lowercase letter i, lowercase letter L, uppercase letter i, and the number one)
- `o`, `O`, and `0` (uppercase letter o, lowercase letter o, and zero)

Basic usage

Creating users

The most basic way to create users is to use the `create_user()` helper function that comes with Django:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword')

# At this point, user is a User object that has already been saved
# to the database. You can continue to change its attributes
# if you want to change other fields.
>>> user.is_staff = True
>>> user.save()
```

You can also create users using the Django admin site. Assuming you've enabled the admin site and hooked it to the URL `/admin/`, the "Add user" page is at `/admin/auth/user/add/`. You should also see a link to "Users" in the "Auth" section of the main admin index page. The "Add user" admin page is different than standard admin pages in that it requires you to choose a username and password before allowing you to edit the rest of the user's fields.

Also note: if you want your own user account to be able to create users using the Django admin site, you'll need to give yourself permission to add users *and* change users (i.e., the "Add user" and "Change user" permissions). If your account has permission to add users but not to change them, you won't be able to add users. Why? Because if you have permission to add users, you have the power to create superusers, which can then, in turn, change other users. So Django requires add *and* change permissions as a slight security measure.

Changing passwords

`manage.py changepassword *username*` offers a method of changing a User's password from the command line. It prompts you to change the password of a given user which you must enter twice. If they both match, the new password will be changed immediately. If you do not supply a user, the command will attempt to change the password whose username matches the current user.

You can also change a password programmatically, using `set_password()`:

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username__exact='john')
>>> u.set_password('new password')
>>> u.save()
```

Don't set the `password` attribute directly unless you know what you're doing. This is explained in the next section.

How Django stores passwords

New in version 1.4: Django 1.4 introduces a new flexible password storage system and uses PBKDF2 by default. Previous versions of Django used SHA1, and other algorithms couldn't be chosen. The `password` attribute of a `User` object is a string in this format:

```
algorithm$hash
```

That's a storage algorithm, and hash, separated by the dollar-sign character. The algorithm is one of a number of one way hashing or password storage algorithms Django can use; see below. The hash is the result of the one- way function.

By default, Django uses the PBKDF2 algorithm with a SHA256 hash, a password stretching mechanism recommended by NIST. This should be sufficient for most users: it's quite secure, requiring massive amounts of computing time to break.

However, depending on your requirements, you may choose a different algorithm, or even use a custom algorithm to match your specific security situation. Again, most users shouldn't need to do this – if you're not sure, you probably don't. If you do, please read on:

Django chooses the an algorithm by consulting the `PASSWORD_HASHERS` setting. This is a list of hashing algorithm classes that this Django installation supports. The first entry in this list (that is, `settings.PASSWORD_HASHERS[0]`) will be used to store passwords, and all the other entries are valid hashers that can be used to check existing passwords. This means that if you want to use a different algorithm, you'll need to modify `PASSWORD_HASHERS` to list your prefered algorithm first in the list.

The default for `PASSWORD_HASHERS` is:

```
PASSWORD_HASHERS = (
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.BCryptPasswordHasher',
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
)
```

This means that Django will use PBKDF2 to store all passwords, but will support checking passwords stored with PBKDF2SHA1, bcrypt, SHA1, etc. The next few sections describe a couple of common ways advanced users may want to modify this setting.

Using bcrypt with Django

Bcrypt is a popular password storage algorithm that's specifically designed for long-term password storage. It's not the default used by Django since it requires the use of third-party libraries, but since many people may want to use it Django supports bcrypt with minimal effort.

To use Bcrypt as your default storage algorithm, do the following:

1. Install the py-bcrypt library (probably by running `sudo pip install py-bcrypt`, or downloading the library and installing it with `python setup.py install`).
2. Modify `PASSWORD_HASHERS` to list `BCryptPasswordHasher` first. That is, in your settings file, you'd put:

   ```
   PASSWORD_HASHERS = (
       'django.contrib.auth.hashers.BCryptPasswordHasher',
       'django.contrib.auth.hashers.PBKDF2PasswordHasher',
       'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
   ```

```
    'django.contrib.auth.hashers.SHA1PasswordHasher',
    'django.contrib.auth.hashers.MD5PasswordHasher',
    'django.contrib.auth.hashers.CryptPasswordHasher',
)
```

(You need to keep the other entries in this list, or else Django won't be able to upgrade passwords; see below).

That's it – now your Django install will use Bcrypt as the default storage algorithm.

Other bcrypt implementations

There are several other implementations that allow bcrypt to be used with Django. Django's bcrypt support is NOT directly compatible with these. To upgrade, you will need to modify the hashes in your database to be in the form *bcrypt$(raw bcrypt output)*. For example: *bcrypt$$2a12NT0I31Sa7ihGEWpka9ASYrEFkhuTNeBQ2xfZskIiiJeyFXhRgS.Sy*.

Increasing the work factor

The PDKDF2 and bcrypt algorithms use a number of iterations or rounds of hashing. This deliberately slows down attackers, making attacks against hashed passwords harder. However, as computing power increases, the number of iterations needs to be increased. We've chosen a reasonable default (and will increase it with each release of Django), but you may wish to tune it up or down, depending on your security needs and available processing power. To do so, you'll subclass the appropriate algorithm and override the `iterations` parameters. For example, to increase the number of iterations used by the default PDKDF2 algorithm:

1. Create a subclass of `django.contrib.auth.hashers.PBKDF2PasswordHasher`:

   ```
   from django.contrib.auth.hashers import PBKDF2PasswordHasher

   class MyPBKDF2PasswordHasher(PBKDF2PasswordHasher):
       """
       A subclass of PBKDF2PasswordHasher that uses 100 times more iterations.
       """
       iterations = PBKDF2PasswordHasher.iterations * 100
   ```

   Save this somewhere in your project. For example, you might put this in a file like `myproject/hashers.py`.

2. Add your new hasher as the first entry in `PASSWORD_HASHERS`:

   ```
   PASSWORD_HASHERS = (
       'myproject.hashers.MyPBKDF2PasswordHasher',
       'django.contrib.auth.hashers.PBKDF2PasswordHasher',
       'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
       'django.contrib.auth.hashers.BCryptPasswordHasher',
       'django.contrib.auth.hashers.SHA1PasswordHasher',
       'django.contrib.auth.hashers.MD5PasswordHasher',
       'django.contrib.auth.hashers.CryptPasswordHasher',
   )
   ```

That's it – now your Django install will use more iterations when it stores passwords using PBKDF2.

Password upgrading

When users log in, if their passwords are stored with anything other than the preferred algorithm, Django will automatically upgrade the algorithm to the preferred one. This means that old installs of Django will get automatically

more secure as users log in, and it also means that you can switch to new (and better) storage algorithms as they get invented.

However, Django can only upgrade passwords that use algorithms mentioned in PASSWORD_HASHERS, so as you upgrade to new systems you should make sure never to *remove* entries from this list. If you do, users using unmentioned algorithms won't be able to upgrade.

Anonymous users

class models.**AnonymousUser**

django.contrib.auth.models.AnonymousUser is a class that implements the django.contrib.auth.models.User interface, with these differences:

- id is always None.
- is_staff and is_superuser are always False.
- is_active is always False.
- groups and user_permissions are always empty.
- is_anonymous() returns True instead of False.
- is_authenticated() returns False instead of True.
- set_password(), check_password(), save(), delete(), set_groups() and set_permissions() raise NotImplementedError.

In practice, you probably won't need to use AnonymousUser objects on your own, but they're used by Web requests, as explained in the next section.

Creating superusers

manage.py syncdb prompts you to create a superuser the first time you run it after adding 'django.contrib.auth' to your INSTALLED_APPS. If you need to create a superuser at a later date, you can use a command line utility:

```
manage.py createsuperuser --username=joe --email=joe@example.com
```

You will be prompted for a password. After you enter one, the user will be created immediately. If you leave off the --username or the --email options, it will prompt you for those values.

If you're using an older release of Django, the old way of creating a superuser on the command line still works:

```
python /path/to/django/contrib/auth/create_superuser.py
```

...where /path/to is the path to the Django codebase on your filesystem. The manage.py command is preferred because it figures out the correct path and environment for you.

Storing additional information about users

If you'd like to store additional information related to your users, Django provides a method to specify a site-specific related model – termed a "user profile" – for this purpose.

To make use of this feature, define a model with fields for the additional information you'd like to store, or additional methods you'd like to have available, and also add a OneToOneField named user from your model to the User model. This will ensure only one instance of your model can be created for each User. For example:

```
from django.contrib.auth.models import User

class UserProfile(models.Model):
    # This field is required.
    user = models.OneToOneField(User)

    # Other fields here
    accepted_eula = models.BooleanField()
    favorite_animal = models.CharField(max_length=20, default="Dragons.")
```

To indicate that this model is the user profile model for a given site, fill in the setting `AUTH_PROFILE_MODULE` with a string consisting of the following items, separated by a dot:

1. The name of the application (case sensitive) in which the user profile model is defined (in other words, the name which was passed to `manage.py startapp` to create the application).
2. The name of the model (not case sensitive) class.

For example, if the profile model was a class named `UserProfile` and was defined inside an application named `accounts`, the appropriate setting would be:

```
AUTH_PROFILE_MODULE = 'accounts.UserProfile'
```

When a user profile model has been defined and specified in this manner, each `User` object will have a method – `get_profile()` – which returns the instance of the user profile model associated with that `User`.

The method `get_profile()` does not create a profile if one does not exist. You need to register a handler for the User model's `django.db.models.signals.post_save` signal and, in the handler, if `created` is `True`, create the associated user profile:

```
# in models.py

from django.contrib.auth.models import User
from django.db.models.signals import post_save

# definition of UserProfile from above
# ...

def create_user_profile(sender, instance, created, **kwargs):
    if created:
        UserProfile.objects.create(user=instance)

post_save.connect(create_user_profile, sender=User)
```

See Also:

Signals for more information on Django's signal dispatcher.

3.9.4 Authentication in Web requests

Until now, this document has dealt with the low-level APIs for manipulating authentication-related objects. On a higher level, Django can hook this authentication framework into its system of `request objects`.

First, install the `SessionMiddleware` and `AuthenticationMiddleware` middlewares by adding them to your `MIDDLEWARE_CLASSES` setting. See the *session documentation* for more information.

Once you have those middlewares installed, you'll be able to access `request.user` in views. `request.user` will give you a `User` object representing the currently logged-in user. If a user isn't currently logged in,

`request.user` will be set to an instance of `AnonymousUser` (see the previous section). You can tell them apart with `is_authenticated()`, like so:

```
if request.user.is_authenticated():
    # Do something for authenticated users.
else:
    # Do something for anonymous users.
```

How to log a user in

Django provides two functions in `django.contrib.auth`: `authenticate()` and `login()`.

`authenticate``()`

To authenticate a given username and password, use `authenticate()`. It takes two keyword arguments, `username` and `password`, and it returns a `User` object if the password is valid for the given username. If the password is invalid, `authenticate()` returns `None`. Example:

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    if user.is_active:
        print("You provided a correct username and password!")
    else:
        print("Your account has been disabled!")
else:
    print("Your username and password were incorrect.")
```

`login``()`

To log a user in, in a view, use `login()`. It takes an `HttpRequest` object and a `User` object. `login()` saves the user's ID in the session, using Django's session framework, so, as mentioned above, you'll need to make sure to have the session middleware installed.

Note that data set during the anonymous session is retained when the user logs in.

This example shows how you might use both `authenticate()` and `login()`:

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        if user.is_active:
            login(request, user)
            # Redirect to a success page.
        else:
            # Return a 'disabled account' error message
    else:
        # Return an 'invalid login' error message.
```

Calling `authenticate()` first

When you're manually logging a user in, you *must* call `authenticate()` before you call `login()`. `authenticate()` sets an attribute on the `User` noting which authentication backend successfully authenticated that user (see the backends documentation for details), and this information is needed later during the login process.

Manually managing a user's password

New in version 1.4: The `django.contrib.auth.hashers` module provides a set of functions to create and validate hashed password. You can use them independently from the `User` model.

check_password(*password*, *encoded*)

New in version 1.4: *Please see the release notes* If you'd like to manually authenticate a user by comparing a plain-text password to the hashed password in the database, use the convenience function `django.contrib.auth.hashers.check_password()`. It takes two arguments: the plain-text password to check, and the full value of a user's `password` field in the database to check against, and returns `True` if they match, `False` otherwise.

make_password(*password*[, *salt*, *hashers*])

New in version 1.4: *Please see the release notes* Creates a hashed password in the format used by this application. It takes one mandatory argument: the password in plain-text. Optionally, you can provide a salt and a hashing algorithm to use, if you don't want to use the defaults (first entry of `PASSWORD_HASHERS` setting). Currently supported algorithms are: `'pbkdf2_sha256'`, `'pbkdf2_sha1'`, `'bcrypt'` (see *Using bcrypt with Django*), `'sha1'`, `'md5'`, `'unsalted_md5'` (only for backward compatibility) and `'crypt'` if you have the `crypt` library installed. If the password argument is `None`, an unusable password is returned (a one that will be never accepted by `django.contrib.auth.hashers.check_password()`).

is_password_usable(*encoded_password*)

New in version 1.4: *Please see the release notes* Checks if the given string is a hashed password that has a chance of being verified against `django.contrib.auth.hashers.check_password()`.

How to log a user out

logout()

To log out a user who has been logged in via `django.contrib.auth.login()`, use `django.contrib.auth.logout()` within your view. It takes an `HttpRequest` object and has no return value. Example:

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Redirect to a success page.
```

Note that `logout()` doesn't throw any errors if the user wasn't logged in.

When you call `logout()`, the session data for the current request is completely cleaned out. All existing data is removed. This is to prevent another person from using the same Web browser to log in and have access to the previous user's session data. If you want to put anything into the session that will be available to the user immediately after logging out, do that *after* calling `django.contrib.auth.logout()`.

Login and logout signals

New in version 1.3: *Please see the release notes* The auth framework uses two *signals* that can be used for notification when a user logs in or out.

`django.contrib.auth.signals.`**user_logged_in**

Sent when a user logs in successfully.

Arguments sent with this signal:

sender As above: the class of the user that just logged in.

request The current `HttpRequest` instance.

user The user instance that just logged in.

`django.contrib.auth.signals.`**user_logged_out**

Sent when the logout method is called.

sender As above: the class of the user that just logged out or `None` if the user was not authenticated.

request The current `HttpRequest` instance.

user The user instance that just logged out or `None` if the user was not authenticated.

Limiting access to logged-in users

The raw way

The simple, raw way to limit access to pages is to check `request.user.is_authenticated()` and either redirect to a login page:

```
from django.http import HttpResponseRedirect

def my_view(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/login/?next=%s' % request.path)
    # ...
```

...or display an error message:

```
def my_view(request):
    if not request.user.is_authenticated():
        return render_to_response('myapp/login_error.html')
    # ...
```

The login_required decorator

`decorators.`**login_required**(\[*redirect_field_name=REDIRECT_FIELD_NAME*, *login_url=None*\])

As a shortcut, you can use the convenient `login_required()` decorator:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...
```

`login_required()` does the following:

- If the user isn't logged in, redirect to `settings.LOGIN_URL`, passing the current absolute path in the query string. Example: `/accounts/login/?next=/polls/3/`.
- If the user is logged in, execute the view normally. The view code is free to assume the user is logged in.

By default, the path that the user should be redirected to upon successful authentication is stored in a query string parameter called `"next"`. If you would prefer to use a different name for this parameter, `login_required()` takes an optional `redirect_field_name` parameter:

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='my_redirect_field')
def my_view(request):
    ...
```

Note that if you provide a value to `redirect_field_name`, you will most likely need to customize your login template as well, since the template context variable which stores the redirect path will use the value of `redirect_field_name` as its key rather than `"next"` (the default). New in version 1.3: *Please see the release notes* `login_required()` also takes an optional `login_url` parameter. Example:

```
from django.contrib.auth.decorators import login_required

@login_required(login_url='/accounts/login/')
def my_view(request):
    ...
```

Note that if you don't specify the `login_url` parameter, you'll need to map the appropriate Django view to `settings.LOGIN_URL`. For example, using the defaults, add the following line to your URLconf:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login'),
```

`views.`**`login`**(*request*[, *template_name*, *redirect_field_name*, *authentication_form*])

URL name: `login`

See *the URL documentation* for details on using named URL patterns.

Here's what `django.contrib.auth.views.login` does:

- If called via `GET`, it displays a login form that POSTs to the same URL. More on this in a bit.
- If called via `POST`, it tries to log the user in. If login is successful, the view redirects to the URL specified in `next`. If `next` isn't provided, it redirects to `settings.LOGIN_REDIRECT_URL` (which defaults to `/accounts/profile/`). If login isn't successful, it redisplays the login form.

It's your responsibility to provide the login form in a template called `registration/login.html` by default. This template gets passed four template context variables:

- `form`: A `Form` object representing the login form. See the *forms documentation* for more on `Form` objects.
- `next`: The URL to redirect to after successful login. This may contain a query string, too.
- `site`: The current `Site`, according to the `SITE_ID` setting. If you don't have the site framework installed, this will be set to an instance of `RequestSite`, which derives the site name and domain from the current `HttpRequest`.
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of `request.META['SERVER_NAME']`. For more on sites, see *The "sites" framework*.

If you'd prefer not to call the template `registration/login.html`, you can pass the `template_name` parameter via the extra arguments to the view in your URLconf. For example, this URLconf line would use `myapp/login.html` instead:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login', {'template_name': 'myapp/login.html'})
```

You can also specify the name of the `GET` field which contains the URL to redirect to after login by passing `redirect_field_name` to the view. By default, the field is called `next`.

Here's a sample `registration/login.html` template you can use as a starting point. It assumes you have a `base.html` template that defines a `content` block:

```
{% extends "base.html" %}

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

<form method="post" action="{% url 'django.contrib.auth.views.login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{% endblock %}
```

If you are using alternate authentication (see *Other authentication sources*) you can pass a custom authentication form to the login view via the `authentication_form` parameter. This form must accept a `request` keyword argument in its `__init__` method, and provide a `get_user` method which returns the authenticated user object (this method is only ever called after successful form validation). New in version 1.4: *Please see the release notes* The `login()` view and the *Other built-in views* now all return a `TemplateResponse` instance, which allows you to easily customize the response data before rendering. For more details, see the *TemplateResponse documentation*.

Other built-in views

In addition to the `login()` view, the authentication system includes a few other useful built-in views located in `django.contrib.auth.views`:

logout(*request*[, *next_page*, *template_name*, *redirect_field_name*])

Logs a user out.

URL name: `logout`

See *the URL documentation* for details on using named URL patterns.

Optional arguments:

- `next_page`: The URL to redirect to after logout.
- `template_name`: The full name of a template to display after logging the user out. Defaults to `registration/logged_out.html` if no argument is supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Overrides `next_page` if the given `GET` parameter is passed.

Template context:

- `title`: The string "Logged out", localized.

- `site`: The current `Site`, according to the `SITE_ID` setting. If you don't have the site framework installed, this will be set to an instance of `RequestSite`, which derives the site name and domain from the current `HttpRequest`.
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of `request.META['SERVER_NAME']`. For more on sites, see *The "sites" framework*.

`logout_then_login`(*request*[, *login_url*])
Logs a user out, then redirects to the login page.

URL name: No default URL provided

Optional arguments:

- `login_url`: The URL of the login page to redirect to. Defaults to `settings.LOGIN_URL` if not supplied.

`password_change`(*request*[, *template_name*, *post_change_redirect*, *password_change_form*])
Allows a user to change their password.

URL name: `password_change`

Optional arguments:

- `template_name`: The full name of a template to use for displaying the password change form. Defaults to `registration/password_change_form.html` if not supplied.
- `post_change_redirect`: The URL to redirect to after a successful password change.
- `password_change_form`: A custom "change password" form which must accept a `user` keyword argument. The form is responsible for actually changing the user's password. Defaults to `PasswordChangeForm`.

Template context:

- `form`: The password change form (see `password_change_form` above).

`password_change_done`(*request*[, *template_name*])
The page shown after a user has changed their password.

URL name: `password_change_done`

Optional arguments:

- `template_name`: The full name of a template to use. Defaults to `registration/password_change_done.html` if not supplied.

`password_reset`(*request*[, *is_admin_site*, *template_name*, *email_template_name*, *password_reset_form*, *token_generator*, *post_reset_redirect*, *from_email*])
Allows a user to reset their password by generating a one-time use link that can be used to reset the password, and sending that link to the user's registered email address. Changed in version 1.3: The `from_email` argument was added.Changed in version 1.4: Users flagged with an unusable password (see `set_unusable_password()` will not be able to request a password reset to prevent misuse when using an external authentication source like LDAP. **URL name:** `password_reset`

Optional arguments:

- `template_name`: The full name of a template to use for displaying the password reset form. Defaults to `registration/password_reset_form.html` if not supplied.
- `email_template_name`: The full name of a template to use for generating the email with the new password. Defaults to `registration/password_reset_email.html` if not supplied.

- `subject_template_name`: The full name of a template to use for the subject of the email with the new password. Defaults to `registration/password_reset_subject.txt` if not supplied. New in version 1.4: *Please see the release notes*
- `password_reset_form`: Form that will be used to set the password. Defaults to `PasswordResetForm`.
- `token_generator`: Instance of the class to check the password. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `post_reset_redirect`: The URL to redirect to after a successful password change.
- `from_email`: A valid email address. By default Django uses the `DEFAULT_FROM_EMAIL`.

Template context:

- `form`: The form (see `password_reset_form` above) for resetting the user's password.

Email template context:

- `email`: An alias for `user.email`
- `user`: The current `User`, according to the `email` form field. Only active users are able to reset their passwords (`User.is_active is True`).
- `site_name`: An alias for `site.name`. If you don't have the site framework installed, this will be set to the value of `request.META['SERVER_NAME']`. For more on sites, see *The "sites" framework*.
- `domain`: An alias for `site.domain`. If you don't have the site framework installed, this will be set to the value of `request.get_host()`.
- `protocol`: http or https
- `uid`: The user's id encoded in base 36.
- `token`: Token to check that the password is valid.

Sample `registration/password_reset_email.html` (email body template):

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' uidb36=uid token=token %}
```

The same template context is used for subject template. Subject must be single line plain text string.

password_reset_done(*request*[, *template_name*])

The page shown after a user has been emailed a link to reset their password. This view is called by default if the `password_reset()` view doesn't have an explicit `post_reset_redirect` URL set.

URL name: `password_reset_done`

Optional arguments:

- `template_name`: The full name of a template to use. Defaults to `registration/password_reset_done.html` if not supplied.

password_reset_confirm(*request*[, *uidb36*, *token*, *template_name*, *token_generator*, *set_password_form*, *post_reset_redirect*])

Presents a form for entering a new password.

URL name: `password_reset_confirm`

Optional arguments:

- `uidb36`: The user's id encoded in base 36. Defaults to `None`.
- `token`: Token to check that the password is valid. Defaults to `None`.

- `template_name`: The full name of a template to display the confirm password view. Default value is `registration/password_reset_confirm.html`.
- `token_generator`: Instance of the class to check the password. This will default to `default_token_generator`, it's an instance of `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `set_password_form`: Form that will be used to set the password. Defaults to `SetPasswordForm`
- `post_reset_redirect`: URL to redirect after the password reset done. Defaults to `None`.

Template context:

- `form`: The form (see `set_password_form` above) for setting the new user's password.
- `validlink`: Boolean, True if the link (combination of uidb36 and token) is valid or unused yet.

password_reset_complete(*request*[, *template_name*])

Presents a view which informs the user that the password has been successfully changed.

URL name: `password_reset_complete`

Optional arguments:

- `template_name`: The full name of a template to display the view. Defaults to `registration/password_reset_complete.html`.

Helper functions

redirect_to_login(*next*[, *login_url*, *redirect_field_name*])

Redirects to the login page, and then back to another URL after a successful login.

Required arguments:

- `next`: The URL to redirect to after a successful login.

Optional arguments:

- `login_url`: The URL of the login page to redirect to. Defaults to `settings.LOGIN_URL` if not supplied.
- `redirect_field_name`: The name of a `GET` field containing the URL to redirect to after log out. Overrides `next` if the given `GET` parameter is passed.

Built-in forms

If you don't want to use the built-in views, but want the convenience of not having to write forms for this functionality, the authentication system provides several built-in forms located in `django.contrib.auth.forms`:

class AdminPasswordChangeForm

A form used in the admin interface to change a user's password.

class AuthenticationForm

A form for logging a user in.

class PasswordChangeForm

A form for allowing a user to change their password.

class PasswordResetForm

A form for generating and emailing a one-time use link to reset a user's password.

class SetPasswordForm
A form that lets a user change his/her password without entering the old password.

class UserChangeForm
A form used in the admin interface to change a user's information and permissions.

class UserCreationForm
A form for creating a new user.

Limiting access to logged-in users that pass a test

To limit access based on certain permissions or some other test, you'd do essentially the same thing as described in the previous section.

The simple way is to run your test on `request.user` in the view directly. For example, this view checks to make sure the user is logged in and has the permission `polls.can_vote`:

```
def my_view(request):
    if not request.user.has_perm('polls.can_vote'):
        return HttpResponse("You can't vote in this poll.")
    # ...
```

user_passes_test(*func*[, *login_url=None*])
As a shortcut, you can use the convenient `user_passes_test` decorator:

```
from django.contrib.auth.decorators import user_passes_test

@user_passes_test(lambda u: u.has_perm('polls.can_vote'))
def my_view(request):
    ...
```

We're using this particular test as a relatively simple example. However, if you just want to test whether a permission is available to a user, you can use the `permission_required()` decorator, described later in this document.

`user_passes_test()` takes a required argument: a callable that takes a `User` object and returns `True` if the user is allowed to view the page. Note that `user_passes_test()` does not automatically check that the `User` is not anonymous.

`user_passes_test()` takes an optional `login_url` argument, which lets you specify the URL for your login page (`settings.LOGIN_URL` by default).

For example:

```
from django.contrib.auth.decorators import user_passes_test

@user_passes_test(lambda u: u.has_perm('polls.can_vote'), login_url='/login/')
def my_view(request):
    ...
```

The permission_required decorator

permission_required(*login_url=None*, *raise_exception=False*])
It's a relatively common task to check whether a user has a particular permission. For that reason, Django provides a shortcut for that case: the `permission_required()` decorator. Using this decorator, the earlier example can be written as:

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote')
def my_view(request):
    ...
```

As for the `User.has_perm()` method, permission names take the form "`<app label>.<permission codename>`" (i.e. `polls.can_vote` for a permission on a model in the `polls` application).

Note that `permission_required()` also takes an optional `login_url` parameter. Example:

```
from django.contrib.auth.decorators import permission_required

@permission_required('polls.can_vote', login_url='/loginpage/')
def my_view(request):
    ...
```

As in the `login_required()` decorator, `login_url` defaults to `settings.LOGIN_URL`. Changed in version 1.4: *Please see the release notes* Added `raise_exception` parameter. If given, the decorator will raise `PermissionDenied`, prompting *the 403 (HTTP Forbidden) view* instead of redirecting to the login page.

Limiting access to generic views

To limit access to a *class-based generic view*, decorate the `View.dispatch` method on the class. See *Decorating the class* for details.

3.9.5 Permissions

Django comes with a simple permissions system. It provides a way to assign permissions to specific users and groups of users.

It's used by the Django admin site, but you're welcome to use it in your own code.

The Django admin site uses permissions as follows:

- Access to view the "add" form and add an object is limited to users with the "add" permission for that type of object.
- Access to view the change list, view the "change" form and change an object is limited to users with the "change" permission for that type of object.
- Access to delete an object is limited to users with the "delete" permission for that type of object.

Permissions are set globally per type of object, not per specific object instance. For example, it's possible to say "Mary may change news stories," but it's not currently possible to say "Mary may change news stories, but only the ones she created herself" or "Mary may only change news stories that have a certain status, publication date or ID." The latter functionality is something Django developers are currently discussing.

Default permissions

When `django.contrib.auth` is listed in your `INSTALLED_APPS` setting, it will ensure that three default permissions – add, change and delete – are created for each Django model defined in one of your installed applications.

These permissions will be created when you run `manage.py syncdb`; the first time you run `syncdb` after adding `django.contrib.auth` to `INSTALLED_APPS`, the default permissions will be created for all previously-installed models, as well as for any new models being installed at that time. Afterward, it will create default permissions for new models each time you run `manage.py syncdb`.

Assuming you have an application with an `app_label` `foo` and a model named `Bar`, to test for basic permissions you should use:

- add: `user.has_perm('foo.add_bar')`
- change: `user.has_perm('foo.change_bar')`
- delete: `user.has_perm('foo.delete_bar')`

Custom permissions

To create custom permissions for a given model object, use the `permissions` *model Meta attribute*.

This example Task model creates three custom permissions, i.e., actions users can or cannot do with Task instances, specific to your application:

```
class Task(models.Model):
    ...
    class Meta:
        permissions = (
            ("view_task", "Can see available tasks"),
            ("change_task_status", "Can change the status of tasks"),
            ("close_task", "Can remove a task by setting its status as closed"),
        )
```

The only thing this does is create those extra permissions when you run `manage.py syncdb`. Your code is in charge of checking the value of these permissions when an user is trying to access the functionality provided by the application (viewing tasks, changing the status of tasks, closing tasks.) Continuing the above example, the following checks if a user may view tasks:

```
user.has_perm('app.view_task')
```

API reference

class `models.`**`Permission`**

Fields

`Permission` objects have the following fields:

`Permission.`**`name`**
Required. 50 characters or fewer. Example: `'Can vote'`.

`Permission.`**`content_type`**
Required. A reference to the `django_content_type` database table, which contains a record for each installed Django model.

`Permission.`**`codename`**
Required. 100 characters or fewer. Example: `'can_vote'`.

Methods

`Permission` objects have the standard data-access methods like any other *Django model*.

Programmatically creating permissions

While custom permissions can be defined within a model's `Meta` class, you can also create permissions directly. For example, you can create the `can_publish` permission for a `BlogPost` model in `myapp`:

```
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get(app_label='myapp', model='BlogPost')
permission = Permission.objects.create(codename='can_publish',
                                       name='Can Publish Posts',
                                       content_type=content_type)
```

The permission can then be assigned to a `User` via its `user_permissions` attribute or to a `Group` via its `permissions` attribute.

3.9.6 Authentication data in templates

The currently logged-in user and his/her permissions are made available in the *template context* when you use `RequestContext`.

Technicality

Technically, these variables are only made available in the template context if you use `RequestContext` *and* your `TEMPLATE_CONTEXT_PROCESSORS` setting contains `"django.contrib.auth.context_processors.auth"`, which is default. For more, see the *RequestContext docs*.

Users

When rendering a template `RequestContext`, the currently logged-in user, either a `User` instance or an `AnonymousUser` instance, is stored in the template variable `{{ user }}`:

```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
    <p>Welcome, new user. Please log in.</p>
{% endif %}
```

This template context variable is not available if a `RequestContext` is not being used.

Permissions

The currently logged-in user's permissions are stored in the template variable `{{ perms }}`. This is an instance of `django.contrib.auth.context_processors.PermWrapper`, which is a template-friendly proxy of permissions. Changed in version 1.3: Prior to version 1.3, `PermWrapper` was located in `django.core.context_processors`. In the `{{ perms }}` object, single-attribute lookup is a proxy to

`User.has_module_perms`. This example would display `True` if the logged-in user had any permissions in the `foo` app:

```
{{ perms.foo }}
```

Two-level-attribute lookup is a proxy to `User.has_perm`. This example would display `True` if the logged-in user had the permission `foo.can_vote`:

```
{{ perms.foo.can_vote }}
```

Thus, you can check permissions in template `{% if %}` statements:

```
{% if perms.foo %}
    <p>You have permission to do something in the foo app.</p>
    {% if perms.foo.can_vote %}
        <p>You can vote!</p>
    {% endif %}
    {% if perms.foo.can_drive %}
        <p>You can drive!</p>
    {% endif %}
{% else %}
    <p>You don't have permission to do anything in the foo app.</p>
{% endif %}
```

3.9.7 Groups

Groups are a generic way of categorizing users so you can apply permissions, or some other label, to those users. A user can belong to any number of groups.

A user in a group automatically has the permissions granted to that group. For example, if the group `Site editors` has the permission `can_edit_home_page`, any user in that group will have that permission.

Beyond permissions, groups are a convenient way to categorize users to give them some label, or extended functionality. For example, you could create a group `'Special users'`, and you could write code that could, say, give them access to a members-only portion of your site, or send them members-only email messages.

API reference

class `models.`**Group**

Fields

`Group` objects have the following fields:

`Group.`**name**
: Required. 80 characters or fewer. Any characters are permitted. Example: `'Awesome Users'`.

`Group.`**permissions**
: Many-to-many field to `Permissions`:

```
group.permissions = [permission_list]
group.permissions.add(permission, permission, ...)
group.permissions.remove(permission, permission, ...)
group.permissions.clear()
```

3.9.8 Other authentication sources

The authentication that comes with Django is good enough for most common cases, but you may have the need to hook into another authentication source – that is, another source of usernames and passwords or authentication methods.

For example, your company may already have an LDAP setup that stores a username and password for every employee. It'd be a hassle for both the network administrator and the users themselves if users had separate accounts in LDAP and the Django-based applications.

So, to handle situations like this, the Django authentication system lets you plug in other authentication sources. You can override Django's default database-based scheme, or you can use the default system in tandem with other systems.

See the *authentication backend reference* for information on the authentication backends included with Django.

Specifying authentication backends

Behind the scenes, Django maintains a list of "authentication backends" that it checks for authentication. When somebody calls `django.contrib.auth.authenticate()` – as described in *How to log a user in* above – Django tries authenticating across all of its authentication backends. If the first authentication method fails, Django tries the second one, and so on, until all backends have been attempted.

The list of authentication backends to use is specified in the `AUTHENTICATION_BACKENDS` setting. This should be a tuple of Python path names that point to Python classes that know how to authenticate. These classes can be anywhere on your Python path.

By default, `AUTHENTICATION_BACKENDS` is set to:

```
('django.contrib.auth.backends.ModelBackend',)
```

That's the basic authentication scheme that checks the Django users database.

The order of `AUTHENTICATION_BACKENDS` matters, so if the same username and password is valid in multiple backends, Django will stop processing at the first positive match.

Note: Once a user has authenticated, Django stores which backend was used to authenticate the user in the user's session, and re-uses the same backend for the duration of that session whenever access to the currently authenticated user is needed. This effectively means that authentication sources are cached on a per-session basis, so if you change `AUTHENTICATION_BACKENDS`, you'll need to clear out session data if you need to force users to re-authenticate using different methods. A simple way to do that is simply to execute `Session.objects.all().delete()`.

Writing an authentication backend

An authentication backend is a class that implements two methods: `get_user(user_id)` and `authenticate(**credentials)`.

The `get_user` method takes a `user_id` – which could be a username, database ID or whatever – and returns a `User` object.

The `authenticate` method takes credentials as keyword arguments. Most of the time, it'll just look like this:

```
class MyBackend(object):
    def authenticate(self, username=None, password=None):
        # Check the username/password and return a User.
```

But it could also authenticate a token, like so:

```
class MyBackend(object):
    def authenticate(self, token=None):
        # Check the token and return a User.
```

Either way, `authenticate` should check the credentials it gets, and it should return a `User` object that matches those credentials, if the credentials are valid. If they're not valid, it should return `None`.

The Django admin system is tightly coupled to the Django `User` object described at the beginning of this document. For now, the best way to deal with this is to create a Django `User` object for each user that exists for your backend (e.g., in your LDAP directory, your external SQL database, etc.) You can either write a script to do this in advance, or your `authenticate` method can do it the first time a user logs in.

Here's an example backend that authenticates against a username and password variable defined in your `settings.py` file and creates a Django `User` object the first time a user authenticates:

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend(object):
    """
    Authenticate against the settings ADMIN_LOGIN and ADMIN_PASSWORD.

    Use the login name, and a hash of the password. For example:

    ADMIN_LOGIN = 'admin'
    ADMIN_PASSWORD = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'
    """

    def authenticate(self, username=None, password=None):
        login_valid = (settings.ADMIN_LOGIN == username)
        pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
        if login_valid and pwd_valid:
            try:
                user = User.objects.get(username=username)
            except User.DoesNotExist:
                # Create a new user. Note that we can set password
                # to anything, because it won't be checked; the password
                # from settings.py will.
                user = User(username=username, password='get from settings.py')
                user.is_staff = True
                user.is_superuser = True
                user.save()
            return user
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Handling authorization in custom backends

Custom auth backends can provide their own permissions.

The user model will delegate permission lookup functions (`get_group_permissions()`, `get_all_permissions()`, `has_perm()`, and `has_module_perms()`) to any authentication backend that implements these functions.

The permissions given to the user will be the superset of all permissions returned by all backends. That is, Django grants a permission to a user that any one backend grants.

The simple backend above could implement permissions for the magic admin fairly simply:

```
class SettingsBackend(object):

    # ...

    def has_perm(self, user_obj, perm, obj=None):
        if user_obj.username == settings.ADMIN_LOGIN:
            return True
        else:
            return False
```

This gives full permissions to the user granted access in the above example. Notice that the backend auth functions all take the user object as an argument, and they also accept the same arguments given to the associated `django.contrib.auth.models.User` functions.

A full authorization implementation can be found in django/contrib/auth/backends.py, which is the default backend and queries the `auth_permission` table most of the time.

Authorization for anonymous users

An anonymous user is one that is not authenticated i.e. they have provided no valid authentication details. However, that does not necessarily mean they are not authorized to do anything. At the most basic level, most Web sites authorize anonymous users to browse most of the site, and many allow anonymous posting of comments etc.

Django's permission framework does not have a place to store permissions for anonymous users. However, it has a foundation that allows custom authentication backends to specify authorization for anonymous users. This is especially useful for the authors of re-usable apps, who can delegate all questions of authorization to the auth backend, rather than needing settings, for example, to control anonymous access.

Authorization for inactive users

New in version 1.3: *Please see the release notes* An inactive user is a one that is authenticated but has its attribute `is_active` set to `False`. However this does not mean they are not authorized to do anything. For example they are allowed to activate their account.

The support for anonymous users in the permission system allows for anonymous users to have permissions to do something while inactive authenticated users do not.

Do not forget to test for the `is_active` attribute of the user in your own backend permission methods.

Handling object permissions

Django's permission framework has a foundation for object permissions, though there is no implementation for it in the core. That means that checking for object permissions will always return `False` or an empty list (depending on the check performed).

3.10 Django's cache framework

A fundamental trade-off in dynamic Web sites is, well, they're dynamic. Each time a user requests a page, the Web server makes all sorts of calculations – from database queries to template rendering to business logic – to create the

page that your site's visitor sees. This is a lot more expensive, from a processing-overhead perspective, than your standard read-a-file-off-the-filesystem server arrangement.

For most Web applications, this overhead isn't a big deal. Most Web applications aren't washingtonpost.com or slashdot.org; they're simply small- to medium-sized sites with so-so traffic. But for medium- to high-traffic sites, it's essential to cut as much overhead as possible.

That's where caching comes in.

To cache something is to save the result of an expensive calculation so that you don't have to perform the calculation next time. Here's some pseudocode explaining how this would work for a dynamically generated Web page:

```
given a URL, try finding that page in the cache
if the page is in the cache:
    return the cached page
else:
    generate the page
    save the generated page in the cache (for next time)
    return the generated page
```

Django comes with a robust cache system that lets you save dynamic pages so they don't have to be calculated for each request. For convenience, Django offers different levels of cache granularity: You can cache the output of specific views, you can cache only the pieces that are difficult to produce, or you can cache your entire site.

Django also works well with "upstream" caches, such as Squid and browser-based caches. These are the types of caches that you don't directly control but to which you can provide hints (via HTTP headers) about which parts of your site should be cached, and how.

3.10.1 Setting up the cache

The cache system requires a small amount of setup. Namely, you have to tell it where your cached data should live – whether in a database, on the filesystem or directly in memory. This is an important decision that affects your cache's performance; yes, some cache types are faster than others.

Your cache preference goes in the `CACHES` setting in your settings file. Here's an explanation of all available values for `CACHES`. Changed in version 1.3: The settings used to configure caching changed in Django 1.3. In Django 1.2 and earlier, you used a single string-based `CACHE_BACKEND` setting to configure caches. This has been replaced with the new dictionary-based `CACHES` setting.

Memcached

By far the fastest, most efficient type of cache available to Django, Memcached is an entirely memory-based cache framework originally developed to handle high loads at LiveJournal.com and subsequently open-sourced by Danga Interactive. It is used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.

Memcached runs as a daemon and is allotted a specified amount of RAM. All it does is provide a fast interface for adding, retrieving and deleting arbitrary data in the cache. All data is stored directly in memory, so there's no overhead of database or filesystem usage.

After installing Memcached itself, you'll need to install a memcached binding. There are several python memcached bindings available; the two most common are python-memcached and pylibmc. Changed in version 1.3: Support for `pylibmc` was added. To use Memcached with Django:

- Set `BACKEND` to `django.core.cache.backends.memcached.MemcachedCache` or `django.core.cache.backends.memcached.PyLibMCCache` (depending on your chosen memcached binding)

- Set `LOCATION` to `ip:port` values, where `ip` is the IP address of the Memcached daemon and `port` is the port on which Memcached is running, or to a `unix:path` value, where `path` is the path to a Memcached Unix socket file.

In this example, Memcached is running on localhost (127.0.0.1) port 11211, using the `python-memcached` binding:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

In this example, Memcached is available through a local Unix socket file `/tmp/memcached.sock` using the `python-memcached` binding:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': 'unix:/tmp/memcached.sock',
    }
}
```

One excellent feature of Memcached is its ability to share cache over multiple servers. This means you can run Memcached daemons on multiple machines, and the program will treat the group of machines as a *single* cache, without the need to duplicate cache values on each machine. To take advantage of this feature, include all server addresses in `LOCATION`, either separated by semicolons or as a list.

In this example, the cache is shared over Memcached instances running on IP address 172.19.26.240 and 172.19.26.242, both on port 11211:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
        ]
    }
}
```

In the following example, the cache is shared over Memcached instances running on the IP addresses 172.19.26.240 (port 11211), 172.19.26.242 (port 11212), and 172.19.26.244 (port 11213):

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': [
            '172.19.26.240:11211',
            '172.19.26.242:11211',
            '172.19.26.244:11213',
        ]
    }
}
```

A final point about Memcached is that memory-based caching has one disadvantage: Because the cached data is stored in memory, the data will be lost if your server crashes. Clearly, memory isn't intended for permanent data storage, so don't rely on memory-based caching as your only data storage. Without a doubt, *none* of the Django caching backends should be used for permanent storage – they're all intended to be solutions for caching, not storage – but we point this out here because memory-based caching is particularly temporary.

Database caching

To use a database table as your cache backend, first create a cache table in your database by running this command:

```
python manage.py createcachetable [cache_table_name]
```

...where `[cache_table_name]` is the name of the database table to create. (This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.) This command creates a single table in your database that is in the proper format that Django's database-cache system expects.

Once you've created that database table, set your `BACKEND` setting to `"django.core.cache.backends.db.DatabaseCache"`, and `LOCATION` to `tablename` – the name of the database table. In this example, the cache table's name is `my_cache_table`:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'my_cache_table',
    }
}
```

The database caching backend uses the same database as specified in your settings file. You can't use a different database backend for your cache table.

Database caching works best if you've got a fast, well-indexed database server.

Database caching and multiple databases

If you use database caching with multiple databases, you'll also need to set up routing instructions for your database cache table. For the purposes of routing, the database cache table appears as a model named `CacheEntry`, in an application named `django_cache`. This model won't appear in the models cache, but the model details can be used for routing purposes.

For example, the following router would direct all cache read operations to `cache_slave`, and all write operations to `cache_master`. The cache table will only be synchronized onto `cache_master`:

```
class CacheRouter(object):
    """A router to control all database cache operations"""

    def db_for_read(self, model, **hints):
        "All cache read operations go to the slave"
        if model._meta.app_label in ('django_cache',):
            return 'cache_slave'
        return None

    def db_for_write(self, model, **hints):
        "All cache write operations go to master"
        if model._meta.app_label in ('django_cache',):
            return 'cache_master'
        return None

    def allow_syncdb(self, db, model):
        "Only synchronize the cache model on master"
        if model._meta.app_label in ('django_cache',):
            return db == 'cache_master'
        return None
```

If you don't specify routing directions for the database cache model, the cache backend will use the `default` database.

Of course, if you don't use the database cache backend, you don't need to worry about providing routing instructions for the database cache model.

Filesystem caching

To store cached items on a filesystem, use "django.core.cache.backends.filebased.FileBasedCache" for BACKEND. For example, to store cached data in /var/tmp/django_cache, use this setting:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

If you're on Windows, put the drive letter at the beginning of the path, like this:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': 'c:/foo/bar',
    }
}
```

The directory path should be absolute – that is, it should start at the root of your filesystem. It doesn't matter whether you put a slash at the end of the setting.

Make sure the directory pointed-to by this setting exists and is readable and writable by the system user under which your Web server runs. Continuing the above example, if your server runs as the user apache, make sure the directory /var/tmp/django_cache exists and is readable and writable by the user apache.

Each cache value will be stored as a separate file whose contents are the cache data saved in a serialized ("pickled") format, using Python's pickle module. Each file's name is the cache key, escaped for safe filesystem use.

Local-memory caching

If you want the speed advantages of in-memory caching but don't have the capability of running Memcached, consider the local-memory cache backend. This cache is multi-process and thread-safe. To use it, set BACKEND to "django.core.cache.backends.locmem.LocMemCache". For example:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'unique-snowflake'
    }
}
```

The cache LOCATION is used to identify individual memory stores. If you only have one locmem cache, you can omit the LOCATION; however, if you have more that one local memory cache, you will need to assign a name to at least one of them in order to keep them separate.

Note that each process will have its own private cache instance, which means no cross-process caching is possible. This obviously also means the local memory cache isn't particularly memory-efficient, so it's probably not a good choice for production environments. It's nice for development.

Dummy caching (for development)

Finally, Django comes with a "dummy" cache that doesn't actually cache – it just implements the cache interface without doing anything.

This is useful if you have a production site that uses heavy-duty caching in various places but a development/test environment where you don't want to cache and don't want to have to change your code to special-case the latter. To activate dummy caching, set `BACKEND` like so:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

Using a custom cache backend

While Django includes support for a number of cache backends out-of-the-box, sometimes you might want to use a customized cache backend. To use an external cache backend with Django, use the Python import path as the `BACKEND` of the `CACHES` setting, like so:

```
CACHES = {
    'default': {
        'BACKEND': 'path.to.backend',
    }
}
```

If you're building your own backend, you can use the standard cache backends as reference implementations. You'll find the code in the `django/core/cache/backends/` directory of the Django source.

Note: Without a really compelling reason, such as a host that doesn't support them, you should stick to the cache backends included with Django. They've been well-tested and are easy to use.

Cache arguments

In addition to the defining the engine and name of the each cache backend, each cache backend can be given additional arguments to control caching behavior. These arguments are provided as additional keys in the `CACHES` setting. Valid arguments are as follows:

- `TIMEOUT`: The default timeout, in seconds, to use for the cache. This argument defaults to 300 seconds (5 minutes).
- `OPTIONS`: Any options that should be passed to cache backend. The list options understood by each backend vary with each backend.

  Cache backends that implement their own culling strategy (i.e., the `locmem`, `filesystem` and `database` backends) will honor the following options:

  - `MAX_ENTRIES`: the maximum number of entries allowed in the cache before old values are deleted. This argument defaults to `300`.
  - `CULL_FREQUENCY`: The fraction of entries that are culled when `MAX_ENTRIES` is reached. The actual ratio is `1/CULL_FREQUENCY`, so set `CULL_FREQUENCY`: to `2` to cull half of the entries when `MAX_ENTRIES` is reached.

    A value of `0` for `CULL_FREQUENCY` means that the entire cache will be dumped when `MAX_ENTRIES` is reached. This makes culling *much* faster at the expense of more cache misses.

Cache backends backed by a third-party library will pass their options directly to the underlying cache library. As a result, the list of valid options depends on the library in use.

- KEY_PREFIX: A string that will be automatically included (prepended by default) to all cache keys used by the Django server.

  See the *cache documentation* for more information.

- VERSION: The default version number for cache keys generated by the Django server.

  See the *cache documentation* for more information.

- KEY_FUNCTION A string containing a dotted path to a function that defines how to compose a prefix, version and key into a final cache key.

  See the *cache documentation* for more information.

In this example, a filesystem backend is being configured with a timeout of 60 seconds, and a maximum capacity of 1000 items:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
        'TIMEOUT': 60,
        'OPTIONS': {
            'MAX_ENTRIES': 1000
        }
    }
}
```

Invalid arguments are silently ignored, as are invalid values of known arguments.

3.10.2 The per-site cache

Once the cache is set up, the simplest way to use caching is to cache your entire site. You'll need to add 'django.middleware.cache.UpdateCacheMiddleware' and 'django.middleware.cache.FetchFromCacheMiddleware' to your MIDDLEWARE_CLASSES setting, as in this example:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

Note: No, that's not a typo: the "update" middleware must be first in the list, and the "fetch" middleware must be last. The details are a bit obscure, but see Order of MIDDLEWARE_CLASSES below if you'd like the full story.

Then, add the following required settings to your Django settings file:

- CACHE_MIDDLEWARE_ALIAS – The cache alias to use for storage.
- CACHE_MIDDLEWARE_SECONDS – The number of seconds each page should be cached.
- CACHE_MIDDLEWARE_KEY_PREFIX – If the cache is shared across multiple sites using the same Django installation, set this to the name of the site, or some other string that is unique to this Django instance, to prevent key collisions. Use an empty string if you don't care.

The cache middleware caches GET and HEAD responses with status 200, where the request and response headers allow. Responses to requests for the same URL with different query parameters are considered to be unique pages and are cached separately. Optionally, if the CACHE_MIDDLEWARE_ANONYMOUS_ONLY setting is True, only anonymous requests (i.e., not those made by a logged-in user) will be cached. This is a simple and effective way of disabling caching for any user-specific pages (including Django's admin interface). Note that if you use CACHE_MIDDLEWARE_ANONYMOUS_ONLY, you should make sure you've activated AuthenticationMiddleware. The cache middleware expects that a HEAD request is answered with the same response headers as the corresponding GET request; in which case it can return a cached GET response for HEAD request.

Additionally, the cache middleware automatically sets a few headers in each HttpResponse:

- Sets the Last-Modified header to the current date/time when a fresh (uncached) version of the page is requested.
- Sets the Expires header to the current date/time plus the defined CACHE_MIDDLEWARE_SECONDS.
- Sets the Cache-Control header to give a max age for the page – again, from the CACHE_MIDDLEWARE_SECONDS setting.

See *Middleware* for more on middleware.

If a view sets its own cache expiry time (i.e. it has a max-age section in its Cache-Control header) then the page will be cached until the expiry time, rather than CACHE_MIDDLEWARE_SECONDS. Using the decorators in django.views.decorators.cache you can easily set a view's expiry time (using the cache_control decorator) or disable caching for a view (using the never_cache decorator). See the using other headers section for more on these decorators. If USE_I18N is set to True then the generated cache key will include the name of the active *language* – see also *How Django discovers language preference*). This allows you to easily cache multilingual sites without having to create the cache key yourself. Changed in version 1.4: *Please see the release notes* Cache keys also include the active *language* when USE_L10N is set to True and the *current time zone* when USE_TZ is set to True.

3.10.3 The per-view cache

django.views.decorators.cache.**cache_page**()

A more granular way to use the caching framework is by caching the output of individual views. django.views.decorators.cache defines a cache_page decorator that will automatically cache the view's response for you. It's easy to use:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)
def my_view(request):
    ...
```

cache_page takes a single argument: the cache timeout, in seconds. In the above example, the result of the my_view() view will be cached for 15 minutes. (Note that we've written it as 60 * 15 for the purpose of readability. 60 * 15 will be evaluated to 900 – that is, 15 minutes multiplied by 60 seconds per minute.)

The per-view cache, like the per-site cache, is keyed off of the URL. If multiple URLs point at the same view, each URL will be cached separately. Continuing the my_view example, if your URLconf looks like this:

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

then requests to /foo/1/ and /foo/23/ will be cached separately, as you may expect. But once a particular URL (e.g., /foo/23/) has been requested, subsequent requests to that URL will use the cache.

`cache_page` can also take an optional keyword argument, `cache`, which directs the decorator to use a specific cache (from your `CACHES` setting) when caching view results. By default, the `default` cache will be used, but you can specify any cache you want:

```
@cache_page(60 * 15, cache="special_cache")
def my_view(request):
    ...
```

You can also override the cache prefix on a per-view basis. `cache_page` takes an optional keyword argument, `key_prefix`, which works in the same way as the `CACHE_MIDDLEWARE_KEY_PREFIX` setting for the middleware. It can be used like this:

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request):
    ...
```

The two settings can also be combined. If you specify a `cache` *and* a `key_prefix`, you will get all the settings of the requested cache alias, but with the key_prefix overridden.

Specifying per-view cache in the URLconf

The examples in the previous section have hard-coded the fact that the view is cached, because `cache_page` alters the `my_view` function in place. This approach couples your view to the cache system, which is not ideal for several reasons. For instance, you might want to reuse the view functions on another, cache-less site, or you might want to distribute the views to people who might want to use them without being cached. The solution to these problems is to specify the per-view cache in the URLconf rather than next to the view functions themselves.

Doing so is easy: simply wrap the view function with `cache_page` when you refer to it in the URLconf. Here's the old URLconf from earlier:

```
urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', my_view),
)
```

Here's the same thing, with `my_view` wrapped in `cache_page`:

```
from django.views.decorators.cache import cache_page

urlpatterns = ('',
    (r'^foo/(\d{1,2})/$', cache_page(60 * 15)(my_view)),
)
```

3.10.4 Template fragment caching

If you're after even more control, you can also cache template fragments using the `cache` template tag. To give your template access to this tag, put `{% load cache %}` near the top of your template.

The `{% cache %}` template tag caches the contents of the block for a given amount of time. It takes at least two arguments: the cache timeout, in seconds, and the name to give the cache fragment. For example:

```
{% load cache %}
{% cache 500 sidebar %}
    .. sidebar ..
{% endcache %}
```

Sometimes you might want to cache multiple copies of a fragment depending on some dynamic data that appears inside the fragment. For example, you might want a separate cached copy of the sidebar used in the previous example

for every user of your site. Do this by passing additional arguments to the `{% cache %}` template tag to uniquely identify the cache fragment:

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

It's perfectly fine to specify more than one argument to identify the fragment. Simply pass as many arguments to `{% cache %}` as you need.

If `USE_I18N` is set to `True` the per-site middleware cache will *respect the active language*. For the `cache` template tag you could use one of the *translation-specific variables* available in templates to achieve the same result:

```
{% load i18n %}
{% load cache %}

{% get_current_language as LANGUAGE_CODE %}

{% cache 600 welcome LANGUAGE_CODE %}
    {% trans "Welcome to example.com" %}
{% endcache %}
```

The cache timeout can be a template variable, as long as the template variable resolves to an integer value. For example, if the template variable `my_timeout` is set to the value `600`, then the following two examples are equivalent:

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

This feature is useful in avoiding repetition in templates. You can set the timeout in a variable, in one place, and just reuse that value.

3.10.5 The low-level cache API

Sometimes, caching an entire rendered page doesn't gain you very much and is, in fact, inconvenient overkill.

Perhaps, for instance, your site includes a view whose results depend on several expensive queries, the results of which change at different intervals. In this case, it would not be ideal to use the full-page caching that the per-site or per-view cache strategies offer, because you wouldn't want to cache the entire result (since some of the data changes often), but you'd still want to cache the results that rarely change.

For cases like this, Django exposes a simple, low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth. (Most common Python objects can be pickled; refer to the Python documentation for more information about pickling.)

The cache module, `django.core.cache`, has a `cache` object that's automatically created from the `'default'` entry in the `CACHES` setting:

```
>>> from django.core.cache import cache
```

The basic interface is `set(key, value, timeout)` and `get(key)`:

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

The `timeout` argument is optional and defaults to the `timeout` argument of the `'default'` backend in `CACHES` setting (explained above). It's the number of seconds the value should be stored in the cache.

If the object doesn't exist in the cache, `cache.get()` returns `None`:

```
# Wait 30 seconds for 'my_key' to expire...

>>> cache.get('my_key')
None
```

We advise against storing the literal value `None` in the cache, because you won't be able to distinguish between your stored `None` value and a cache miss signified by a return value of `None`.

`cache.get()` can take a `default` argument. This specifies which value to return if the object doesn't exist in the cache:

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

To add a key only if it doesn't already exist, use the `add()` method. It takes the same parameters as `set()`, but it will not attempt to update the cache if the key specified is already present:

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

If you need to know whether `add()` stored a value in the cache, you can check the return value. It will return `True` if the value was stored, `False` otherwise.

There's also a `get_many()` interface that only hits the cache once. `get_many()` returns a dictionary with all the keys you asked for that actually exist in the cache (and haven't expired):

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

To set multiple values more efficiently, use `set_many()` to pass a dictionary of key-value pairs:

```
>>> cache.set_many({'a': 1, 'b': 2, 'c': 3})
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

Like `cache.set()`, `set_many()` takes an optional `timeout` parameter.

You can delete keys explicitly with `delete()`. This is an easy way of clearing the cache for a particular object:

```
>>> cache.delete('a')
```

If you want to clear a bunch of keys at once, `delete_many()` can take a list of keys to be cleared:

```
>>> cache.delete_many(['a', 'b', 'c'])
```

Finally, if you want to delete all the keys in the cache, use `cache.clear()`. Be careful with this; `clear()` will remove *everything* from the cache, not just the keys set by your application.

```
>>> cache.clear()
```

You can also increment or decrement a key that already exists using the `incr()` or `decr()` methods, respectively. By default, the existing cache value will incremented or decremented by 1. Other increment/decrement values can be specified by providing an argument to the increment/decrement call. A ValueError will be raised if you attempt to increment or decrement a nonexistent cache key.:

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

Note: `incr()`/`decr()` methods are not guaranteed to be atomic. On those backends that support atomic increment/decrement (most notably, the memcached backend), increment and decrement operations will be atomic. However, if the backend doesn't natively provide an increment/decrement operation, it will be implemented using a two-step retrieve/update.

Cache key prefixing

New in version 1.3: *Please see the release notes* If you are sharing a cache instance between servers, or between your production and development environments, it's possible for data cached by one server to be used by another server. If the format of cached data is different between servers, this can lead to some very hard to diagnose problems.

To prevent this, Django provides the ability to prefix all cache keys used by a server. When a particular cache key is saved or retrieved, Django will automatically prefix the cache key with the value of the `KEY_PREFIX` cache setting.

By ensuring each Django instance has a different `KEY_PREFIX`, you can ensure that there will be no collisions in cache values.

Cache versioning

New in version 1.3: *Please see the release notes* When you change running code that uses cached values, you may need to purge any existing cached values. The easiest way to do this is to flush the entire cache, but this can lead to the loss of cache values that are still valid and useful.

Django provides a better way to target individual cache values. Django's cache framework has a system-wide version identifier, specified using the `VERSION` cache setting. The value of this setting is automatically combined with the cache prefix and the user-provided cache key to obtain the final cache key.

By default, any key request will automatically include the site default cache key version. However, the primitive cache functions all include a `version` argument, so you can specify a particular cache key version to set or get. For example:

```
# Set version 2 of a cache key
>>> cache.set('my_key', 'hello world!', version=2)
# Get the default version (assuming version=1)
>>> cache.get('my_key')
None
# Get version 2 of the same key
>>> cache.get('my_key', version=2)
'hello world!'
```

The version of a specific key can be incremented and decremented using the `incr_version()` and `decr_version()` methods. This enables specific keys to be bumped to a new version, leaving other keys unaffected. Continuing our previous example:

```
# Increment the version of 'my_key'
>>> cache.incr_version('my_key')
# The default version still isn't available
>>> cache.get('my_key')
None
# Version 2 isn't available, either
>>> cache.get('my_key', version=2)
None
# But version 3 *is* available
>>> cache.get('my_key', version=3)
'hello world!'
```

Cache key transformation

New in version 1.3: *Please see the release notes* As described in the previous two sections, the cache key provided by a user is not used verbatim – it is combined with the cache prefix and key version to provide a final cache key. By default, the three parts are joined using colons to produce a final string:

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), smart_str(key)])
```

If you want to combine the parts in different ways, or apply other processing to the final key (e.g., taking a hash digest of the key parts), you can provide a custom key function.

The `KEY_FUNCTION` cache setting specifies a dotted-path to a function matching the prototype of `make_key()` above. If provided, this custom key function will be used instead of the default key combining function.

Cache key warnings

New in version 1.3: *Please see the release notes* Memcached, the most commonly-used production cache backend, does not allow cache keys longer than 250 characters or containing whitespace or control characters, and using such keys will cause an exception. To encourage cache-portable code and minimize unpleasant surprises, the other built-in cache backends issue a warning (`django.core.cache.backends.base.CacheKeyWarning`) if a key is used that would cause an error on memcached.

If you are using a production backend that can accept a wider range of keys (a custom backend, or one of the non-memcached built-in backends), and want to use this wider range without warnings, you can silence `CacheKeyWarning` with this code in the `management` module of one of your `INSTALLED_APPS`:

```
import warnings

from django.core.cache import CacheKeyWarning

warnings.simplefilter("ignore", CacheKeyWarning)
```

If you want to instead provide custom key validation logic for one of the built-in backends, you can subclass it, override just the `validate_key` method, and follow the instructions for using a custom cache backend. For instance, to do this for the `locmem` backend, put this code in a module:

```
from django.core.cache.backends.locmem import LocMemCache

class CustomLocMemCache(LocMemCache):
    def validate_key(self, key):
        """Custom validation, raising exceptions or warnings as needed."""
        # ...
```

...and use the dotted Python path to this class in the `BACKEND` portion of your `CACHES` setting.

3.10.6 Upstream caches

So far, this document has focused on caching your *own* data. But another type of caching is relevant to Web development, too: caching performed by "upstream" caches. These are systems that cache pages for users even before the request reaches your Web site.

Here are a few examples of upstream caches:

- Your ISP may cache certain pages, so if you requested a page from http://example.com/, your ISP would send you the page without having to access example.com directly. The maintainers of example.com have no knowledge of this caching; the ISP sits between example.com and your Web browser, handling all of the caching transparently.
- Your Django Web site may sit behind a *proxy cache*, such as Squid Web Proxy Cache (http://www.squid-cache.org/), that caches pages for performance. In this case, each request first would be handled by the proxy, and it would be passed to your application only if needed.
- Your Web browser caches pages, too. If a Web page sends out the appropriate headers, your browser will use the local cached copy for subsequent requests to that page, without even contacting the Web page again to see whether it has changed.

Upstream caching is a nice efficiency boost, but there's a danger to it: Many Web pages' contents differ based on authentication and a host of other variables, and cache systems that blindly save pages based purely on URLs could expose incorrect or sensitive data to subsequent visitors to those pages.

For example, say you operate a Web email system, and the contents of the "inbox" page obviously depend on which user is logged in. If an ISP blindly cached your site, then the first user who logged in through that ISP would have his user-specific inbox page cached for subsequent visitors to the site. That's not cool.

Fortunately, HTTP provides a solution to this problem. A number of HTTP headers exist to instruct upstream caches to differ their cache contents depending on designated variables, and to tell caching mechanisms not to cache particular pages. We'll look at some of these headers in the sections that follow.

3.10.7 Using Vary headers

The `Vary` header defines which request headers a cache mechanism should take into account when building its cache key. For example, if the contents of a Web page depend on a user's language preference, the page is said to "vary on language." Changed in version 1.3: In Django 1.3 the full request path – including the query – is used to create the cache keys, instead of only the path component in Django 1.2. By default, Django's cache system creates its cache keys using the requested path and query – e.g., `"/stories/2005/?order_by=author"`. This means every request to that URL will use the same cached version, regardless of user-agent differences such as cookies or language preferences. However, if this page produces different content based on some difference in request headers – such as a cookie, or a language, or a user-agent – you'll need to use the `Vary` header to tell caching mechanisms that the page output depends on those things.

To do this in Django, use the convenient `vary_on_headers` view decorator, like so:

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers('User-Agent')
def my_view(request):
    # ...
```

In this case, a caching mechanism (such as Django's own cache middleware) will cache a separate version of the page for each unique user-agent.

The advantage to using the `vary_on_headers` decorator rather than manually setting the `Vary` header (using something like `response['Vary'] = 'user-agent'`) is that the decorator *adds* to the `Vary` header (which may already exist), rather than setting it from scratch and potentially overriding anything that was already in there.

You can pass multiple headers to `vary_on_headers()`:

```
@vary_on_headers('User-Agent', 'Cookie')
def my_view(request):
    # ...
```

This tells upstream caches to vary on *both*, which means each combination of user-agent and cookie will get its own cache value. For example, a request with the user-agent `Mozilla` and the cookie value `foo=bar` will be considered different from a request with the user-agent `Mozilla` and the cookie value `foo=ham`.

Because varying on cookie is so common, there's a `vary_on_cookie` decorator. These two views are equivalent:

```
@vary_on_cookie
def my_view(request):
    # ...

@vary_on_headers('Cookie')
def my_view(request):
    # ...
```

The headers you pass to `vary_on_headers` are not case sensitive; `"User-Agent"` is the same thing as `"user-agent"`.

You can also use a helper function, `django.utils.cache.patch_vary_headers`, directly. This function sets, or adds to, the `Vary header`. For example:

```
from django.utils.cache import patch_vary_headers

def my_view(request):
    # ...
    response = render_to_response('template_name', context)
    patch_vary_headers(response, ['Cookie'])
    return response
```

`patch_vary_headers` takes an `HttpResponse` instance as its first argument and a list/tuple of case-insensitive header names as its second argument.

For more on Vary headers, see the official Vary spec.

3.10.8 Controlling cache: Using other headers

Other problems with caching are the privacy of data and the question of where data should be stored in a cascade of caches.

A user usually faces two kinds of caches: his or her own browser cache (a private cache) and his or her provider's cache (a public cache). A public cache is used by multiple users and controlled by someone else. This poses problems with sensitive data–you don't want, say, your bank account number stored in a public cache. So Web applications need a way to tell caches which data is private and which is public.

The solution is to indicate a page's cache should be "private." To do this in Django, use the `cache_control` view decorator. Example:

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def my_view(request):
    # ...
```

This decorator takes care of sending out the appropriate HTTP header behind the scenes.

Note that the cache control settings "private" and "public" are mutually exclusive. The decorator ensures that the "public" directive is removed if "private" should be set (and vice versa). An example use of the two directives would be a blog site that offers both private and public entries. Public entries may be cached on any shared cache. The following code uses `patch_cache_control`, the manual way to modify the cache control header (it is internally called by the `cache_control` decorator):

```
from django.views.decorators.cache import patch_cache_control
from django.views.decorators.vary import vary_on_cookie

@vary_on_cookie
def list_blog_entries_view(request):
    if request.user.is_anonymous():
        response = render_only_public_entries()
        patch_cache_control(response, public=True)
    else:
        response = render_private_and_public_entries(request.user)
        patch_cache_control(response, private=True)

    return response
```

There are a few other ways to control cache parameters. For example, HTTP allows applications to do the following:

- Define the maximum time a page should be cached.
- Specify whether a cache should always check for newer versions, only delivering the cached content when there are no changes. (Some caches might deliver cached content even if the server page changed, simply because the cache copy isn't yet expired.)

In Django, use the `cache_control` view decorator to specify these cache parameters. In this example, `cache_control` tells caches to revalidate the cache on every access and to store cached versions for, at most, 3,600 seconds:

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True, max_age=3600)
def my_view(request):
    # ...
```

Any valid `Cache-Control` HTTP directive is valid in `cache_control()`. Here's a full list:

- `public=True`
- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True`
- `proxy_revalidate=True`
- `max_age=num_seconds`
- `s_maxage=num_seconds`

For explanation of Cache-Control HTTP directives, see the Cache-Control spec.

(Note that the caching middleware already sets the cache header's max-age with the value of the `CACHE_MIDDLEWARE_SECONDS` setting. If you use a custom `max_age` in a `cache_control` decorator, the decorator will take precedence, and the header values will be merged correctly.)

If you want to use headers to disable caching altogether, `django.views.decorators.cache.never_cache` is a view decorator that adds headers to ensure the response won't be cached by browsers or other caches. Example:

```
from django.views.decorators.cache import never_cache

@never_cache
def myview(request):
    # ...
```

3.10.9 Other optimizations

Django comes with a few other pieces of middleware that can help optimize your site's performance:

- `django.middleware.http.ConditionalGetMiddleware` adds support for modern browsers to conditionally GET responses based on the `ETag` and `Last-Modified` headers.
- `django.middleware.gzip.GZipMiddleware` compresses responses for all modern browsers, saving bandwidth and transfer time.

3.10.10 Order of MIDDLEWARE_CLASSES

If you use caching middleware, it's important to put each half in the right place within the `MIDDLEWARE_CLASSES` setting. That's because the cache middleware needs to know which headers by which to vary the cache storage. Middleware always adds something to the `Vary` response header when it can.

`UpdateCacheMiddleware` runs during the response phase, where middleware is run in reverse order, so an item at the top of the list runs *last* during the response phase. Thus, you need to make sure that `UpdateCacheMiddleware` appears *before* any other middleware that might add something to the `Vary` header. The following middleware modules do so:

- `SessionMiddleware` adds `Cookie`
- `GZipMiddleware` adds `Accept-Encoding`
- `LocaleMiddleware` adds `Accept-Language`

`FetchFromCacheMiddleware`, on the other hand, runs during the request phase, where middleware is applied first-to-last, so an item at the top of the list runs *first* during the request phase. The `FetchFromCacheMiddleware` also needs to run after other middleware updates the `Vary` header, so `FetchFromCacheMiddleware` must be *after* any item that does so.

3.11 Conditional View Processing

HTTP clients can send a number of headers to tell the server about copies of a resource that they have already seen. This is commonly used when retrieving a Web page (using an HTTP `GET` request) to avoid sending all the data for something the client has already retrieved. However, the same headers can be used for all HTTP methods (`POST`, `PUT`, `DELETE`, etc).

For each page (response) that Django sends back from a view, it might provide two HTTP headers: the `ETag` header and the `Last-Modified` header. These headers are optional on HTTP responses. They can be set by your view function, or you can rely on the `CommonMiddleware` middleware to set the `ETag` header.

When the client next requests the same resource, it might send along a header such as If-modified-since, containing the date of the last modification time it was sent, or If-none-match, containing the `ETag` it was sent. If the current version of the page matches the `ETag` sent by the client, or if the resource has not been modified, a 304 status code can be sent back, instead of a full response, telling the client that nothing has changed.

When you need more fine-grained control you may use per-view conditional processing functions.

3.11.1 The `condition` decorator

Sometimes (in fact, quite often) you can create functions to rapidly compute the ETag value or the last-modified time for a resource, **without** needing to do all the computations needed to construct the full view. Django can then use these functions to provide an "early bailout" option for the view processing. Telling the client that the content has not been modified since the last request, perhaps.

These two functions are passed as parameters the `django.views.decorators.http.condition` decorator. This decorator uses the two functions (you only need to supply one, if you can't compute both quantities easily and quickly) to work out if the headers in the HTTP request match those on the resource. If they don't match, a new copy of the resource must be computed and your normal view is called.

The `condition` decorator's signature looks like this:

```
condition(etag_func=None, last_modified_func=None)
```

The two functions, to compute the ETag and the last modified time, will be passed the incoming `request` object and the same parameters, in the same order, as the view function they are helping to wrap. The function passed `last_modified_func` should return a standard datetime value specifying the last time the resource was modified, or `None` if the resource doesn't exist. The function passed to the `etag` decorator should return a string representing the Etag for the resource, or `None` if it doesn't exist.

Using this feature usefully is probably best explained with an example. Suppose you have this pair of models, representing a simple blog system:

```
import datetime
from django.db import models

class Blog(models.Model):
    ...

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    published = models.DateTimeField(default=datetime.datetime.now)
    ...
```

If the front page, displaying the latest blog entries, only changes when you add a new blog entry, you can compute the last modified time very quickly. You need the latest `published` date for every entry associated with that blog. One way to do this would be:

```
def latest_entry(request, blog_id):
    return Entry.objects.filter(blog=blog_id).latest("published").published
```

You can then use this function to provide early detection of an unchanged page for your front page view:

```
from django.views.decorators.http import condition

@condition(last_modified_func=latest_entry)
def front_page(request, blog_id):
    ...
```

3.11.2 Shortcuts for only computing one value

As a general rule, if you can provide functions to compute *both* the ETag and the last modified time, you should do so. You don't know which headers any given HTTP client will send you, so be prepared to handle both. However, sometimes only one value is easy to compute and Django provides decorators that handle only ETag or only last-modified computations.

The `django.views.decorators.http.etag` and `django.views.decorators.http.last_modified` decorators are passed the same type of functions as the `condition` decorator. Their signatures are:

```
etag(etag_func)
last_modified(last_modified_func)
```

We could write the earlier example, which only uses a last-modified function, using one of these decorators:

```
@last_modified(latest_entry)
def front_page(request, blog_id):
    ...
```

...or:

```
def front_page(request, blog_id):
    ...
front_page = last_modified(latest_entry)(front_page)
```

Use `condition` when testing both conditions

It might look nicer to some people to try and chain the `etag` and `last_modified` decorators if you want to test both preconditions. However, this would lead to incorrect behavior.

```
# Bad code. Don't do this!
@etag(etag_func)
@last_modified(last_modified_func)
def my_view(request):
    # ...

# End of bad code.
```

The first decorator doesn't know anything about the second and might answer that the response is not modified even if the second decorators would determine otherwise. The `condition` decorator uses both callback functions simultaneously to work out the right action to take.

3.11.3 Using the decorators with other HTTP methods

The `condition` decorator is useful for more than only `GET` and `HEAD` requests (`HEAD` requests are the same as `GET` in this situation). It can be used also to be used to provide checking for `POST`, `PUT` and `DELETE` requests. In these situations, the idea isn't to return a "not modified" response, but to tell the client that the resource they are trying to change has been altered in the meantime.

For example, consider the following exchange between the client and server:

1. Client requests `/foo/`.
2. Server responds with some content with an ETag of `"abcd1234"`.
3. Client sends an HTTP `PUT` request to `/foo/` to update the resource. It also sends an `If-Match: "abcd1234"` header to specify the version it is trying to update.
4. Server checks to see if the resource has changed, by computing the ETag the same way it does for a `GET` request (using the same function). If the resource *has* changed, it will return a 412 status code code, meaning "precondition failed".
5. Client sends a `GET` request to `/foo/`, after receiving a 412 response, to retrieve an updated version of the content before updating it.

The important thing this example shows is that the same functions can be used to compute the ETag and last modification values in all situations. In fact, you **should** use the same functions, so that the same values are returned every time.

3.11.4 Comparison with middleware conditional processing

You may notice that Django already provides simple and straightforward conditional `GET` handling via the `django.middleware.http.ConditionalGetMiddleware` and `CommonMiddleware`. Whilst certainly being easy to use and suitable for many situations, those pieces of middleware functionality have limitations for advanced usage:

- They are applied globally to all views in your project
- They don't save you from generating the response itself, which may be expensive
- They are only appropriate for HTTP `GET` requests.

You should choose the most appropriate tool for your particular problem here. If you have a way to compute ETags and modification times quickly and if some view takes a while to generate the content, you should consider using the `condition` decorator described in this document. If everything already runs fairly quickly, stick to using the middleware and the amount of network traffic sent back to the clients will still be reduced if the view hasn't changed.

3.12 Cryptographic signing

New in version 1.4: *Please see the release notes* The golden rule of Web application security is to never trust data from untrusted sources. Sometimes it can be useful to pass data through an untrusted medium. Cryptographically signed values can be passed through an untrusted channel safe in the knowledge that any tampering will be detected.

Django provides both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in Web applications.

You may also find signing useful for the following:

- Generating "recover my account" URLs for sending to users who have lost their password.
- Ensuring data stored in hidden form fields has not been tampered with.
- Generating one-time secret URLs for allowing temporary access to a protected resource, for example a downloadable file that a user has paid for.

3.12.1 Protecting the SECRET_KEY

When you create a new Django project using `startproject`, the `settings.py` file is generated automatically and gets a random `SECRET_KEY` value. This value is the key to securing signed data – it is vital you keep this secure, or attackers could use it to generate their own signed values.

3.12.2 Using the low-level API

class `Signer`

Django's signing methods live in the `django.core.signing` module. To sign a value, first instantiate a `Signer` instance:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> value = signer.sign('My string')
>>> value
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
```

The signature is appended to the end of the string, following the colon. You can retrieve the original value using the `unsign` method:

```
>>> original = signer.unsign(value)
>>> original
u'My string'
```

If the signature or value have been altered in any way, a `django.core.signing.BadSignature` exception will be raised:

```
>>> value += 'm'
>>> try:
...     original = signer.unsign(value)
... except signing.BadSignature:
...     print("Tampering detected!")
```

By default, the `Signer` class uses the `SECRET_KEY` setting to generate signatures. You can use a different secret by passing it to the `Signer` constructor:

```
>>> signer = Signer('my-other-secret')
>>> value = signer.sign('My string')
>>> value
'My string:EkfQJafvGyiofrdGnuthdxImIJw'
```

Using the salt argument

If you do not wish for every occurrence of a particular string to have the same signature hash, you can use the optional `salt` argument to the `Signer` class. Using a salt will seed the signing hash function with both the salt and your `SECRET_KEY`:

```
>>> signer = Signer()
>>> signer.sign('My string')
'My string:GdMGD6HNQ_qdgxYP8yBZAdAIV1w'
>>> signer = Signer(salt='extra')
>>> signer.sign('My string')
'My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw'
>>> signer.unsign('My string:Ee7vGi-ING6n02gkcJ-QLHg6vFw')
u'My string'
```

Using salt in this way puts the different signatures into different namespaces. A signature that comes from one namespace (a particular salt value) cannot be used to validate the same plaintext string in a different namespace that is using a different salt setting. The result is to prevent an attacker from using a signed string generated in one place in the code as input to another piece of code that is generating (and verifying) signatures using a different salt.

Unlike your `SECRET_KEY`, your salt argument does not need to stay secret.

Verifying timestamped values

class `TimestampSigner`

`TimestampSigner` is a subclass of `Signer` that appends a signed timestamp to the value. This allows you to confirm that a signed value was created within a specified period of time:

```
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> value = signer.sign('hello')
>>> value
'hello:1NMg5H:oPVuCqlJWmChm1rA2lyTUtelC-c'
>>> signer.unsign(value)
u'hello'
>>> signer.unsign(value, max_age=10)
...
SignatureExpired: Signature age 15.5289158821 > 10 seconds
>>> signer.unsign(value, max_age=20)
u'hello'
```

Protecting complex data structures

If you wish to protect a list, tuple or dictionary you can do so using the signing module's `dumps` and `loads` functions. These imitate Python's pickle module, but use JSON serialization under the hood. JSON ensures that even if your `SECRET_KEY` is stolen an attacker will not be able to execute arbitrary commands by exploiting the pickle format.:

```
>>> from django.core import signing
>>> value = signing.dumps({"foo": "bar"})
>>> value
'eyJmb28iOiJiYXIifQ:1NMg1b:zGcDE4-TCkaeGzLeW9UQwZesciI'
>>> signing.loads(value)
{'foo': 'bar'}
```

dumps(*obj*, *key=None*, *salt='django.core.signing'*, *compress=False*)
Returns URL-safe, sha1 signed base64 compressed JSON string.

loads(*string*, *key=None*, *salt='django.core.signing'*, *max_age=None*)
Reverse of dumps(), raises `BadSignature` if signature fails.

3.13 Sending email

Although Python makes sending email relatively easy via the `smtplib` module, Django provides a couple of light wrappers over it. These wrappers are provided to make sending email extra quick, to make it easy to test email sending during development, and to provide support for platforms that can't use SMTP.

The code lives in the `django.core.mail` module.

3.13.1 Quick example

In two lines:

```
from django.core.mail import send_mail

send_mail('Subject here', 'Here is the message.', 'from@example.com',
    ['to@example.com'], fail_silently=False)
```

Mail is sent using the SMTP host and port specified in the `EMAIL_HOST` and `EMAIL_PORT` settings. The `EMAIL_HOST_USER` and `EMAIL_HOST_PASSWORD` settings, if set, are used to authenticate to the SMTP server, and the `EMAIL_USE_TLS` setting controls whether a secure connection is used.

Note: The character set of email sent with `django.core.mail` will be set to the value of your `DEFAULT_CHARSET` setting.

3.13.2 send_mail()

send_mail(*subject*, *message*, *from_email*, *recipient_list*, *fail_silently=False*, *auth_user=None*, *auth_password=None*, *connection=None*)

The simplest way to send email is using `django.core.mail.send_mail()`.

The `subject`, `message`, `from_email` and `recipient_list` parameters are required.

- `subject`: A string.
- `message`: A string.
- `from_email`: A string.
- `recipient_list`: A list of strings, each an email address. Each member of `recipient_list` will see the other recipients in the "To:" field of the email message.
- `fail_silently`: A boolean. If it's `False`, `send_mail` will raise an `smtplib.SMTPException`. See the `smtplib` docs for a list of possible exceptions, all of which are subclasses of `SMTPException`.
- `auth_user`: The optional username to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the `EMAIL_HOST_USER` setting.
- `auth_password`: The optional password to use to authenticate to the SMTP server. If this isn't provided, Django will use the value of the `EMAIL_HOST_PASSWORD` setting.
- `connection`: The optional email backend to use to send the mail. If unspecified, an instance of the default backend will be used. See the documentation on *Email backends* for more details.

3.13.3 send_mass_mail()

send_mass_mail(*datatuple*, *fail_silently=False*, *auth_user=None*, *auth_password=None*, *connection=None*)

`django.core.mail.send_mass_mail()` is intended to handle mass emailing.

`datatuple` is a tuple in which each element is in this format:

```
(subject, message, from_email, recipient_list)
```

`fail_silently`, `auth_user` and `auth_password` have the same functions as in `send_mail()`.

Each separate element of `datatuple` results in a separate email message. As in `send_mail()`, recipients in the same `recipient_list` will all see the other addresses in the email messages' "To:" field.

For example, the following code would send two different messages to two different sets of recipients; however, only one connection to the mail server would be opened:

```
message1 = ('Subject here', 'Here is the message', 'from@example.com', ['first@example.com', 'other@e
message2 = ('Another Subject', 'Here is another message', 'from@example.com', ['second@test.com'])
send_mass_mail((message1, message2), fail_silently=False)
```

send_mass_mail() vs. send_mail()

The main difference between `send_mass_mail()` and `send_mail()` is that `send_mail()` opens a connection to the mail server each time it's executed, while `send_mass_mail()` uses a single connection for all of its messages. This makes `send_mass_mail()` slightly more efficient.

3.13.4 mail_admins()

mail_admins(*subject*, *message*, *fail_silently=False*, *connection=None*, *html_message=None*)

`django.core.mail.mail_admins()` is a shortcut for sending an email to the site admins, as defined in the `ADMINS` setting.

`mail_admins()` prefixes the subject with the value of the `EMAIL_SUBJECT_PREFIX` setting, which is `"[Django] "` by default.

The "From:" header of the email will be the value of the `SERVER_EMAIL` setting.

This method exists for convenience and readability. Changed in version 1.3: *Please see the release notes* If `html_message` is provided, the resulting email will be a *`multipart/alternative`* email with `message` as the *`text/plain`* content type and `html_message` as the *`text/html`* content type.

3.13.5 mail_managers()

mail_managers(*subject*, *message*, *fail_silently=False*, *connection=None*, *html_message=None*)

`django.core.mail.mail_managers()` is just like `mail_admins()`, except it sends an email to the site managers, as defined in the `MANAGERS` setting.

3.13.6 Examples

This sends a single email to john@example.com and jane@example.com, with them both appearing in the "To:":

```
send_mail('Subject', 'Message.', 'from@example.com',
    ['john@example.com', 'jane@example.com'])
```

This sends a message to john@example.com and jane@example.com, with them both receiving a separate email:

```
datatuple = (
    ('Subject', 'Message.', 'from@example.com', ['john@example.com']),
    ('Subject', 'Message.', 'from@example.com', ['jane@example.com']),
)
send_mass_mail(datatuple)
```

3.13.7 Preventing header injection

Header injection is a security exploit in which an attacker inserts extra email headers to control the "To:" and "From:" in email messages that your scripts generate.

The Django email functions outlined above all protect against header injection by forbidding newlines in header values. If any `subject`, `from_email` or `recipient_list` contains a newline (in either Unix, Windows or Mac style), the email function (e.g. `send_mail()`) will raise `django.core.mail.BadHeaderError` (a subclass of `ValueError`) and, hence, will not send the email. It's your responsibility to validate all data before passing it to the email functions.

If a `message` contains headers at the start of the string, the headers will simply be printed as the first bit of the email message.

Here's an example view that takes a `subject`, `message` and `from_email` from the request's POST data, sends that to admin@example.com and redirects to "/contact/thanks/" when it's done:

```
from django.core.mail import send_mail, BadHeaderError

def send_email(request):
    subject = request.POST.get('subject', '')
    message = request.POST.get('message', '')
    from_email = request.POST.get('from_email', '')
    if subject and message and from_email:
        try:
            send_mail(subject, message, from_email, ['admin@example.com'])
        except BadHeaderError:
            return HttpResponse('Invalid header found.')
        return HttpResponseRedirect('/contact/thanks/')
    else:
        # In reality we'd use a form class
        # to get proper validation errors.
        return HttpResponse('Make sure all fields are entered and valid.')
```

3.13.8 The EmailMessage class

Django's `send_mail()` and `send_mass_mail()` functions are actually thin wrappers that make use of the `EmailMessage` class.

Not all features of the `EmailMessage` class are available through the `send_mail()` and related wrapper functions. If you wish to use advanced features, such as BCC'ed recipients, file attachments, or multi-part email, you'll need to create `EmailMessage` instances directly.

Note: This is a design feature. `send_mail()` and related functions were originally the only interface Django provided. However, the list of parameters they accepted was slowly growing over time. It made sense to move to a more object-oriented design for email messages and retain the original functions only for backwards compatibility.

`EmailMessage` is responsible for creating the email message itself. The *email backend* is then responsible for sending the email.

For convenience, `EmailMessage` provides a simple `send()` method for sending a single email. If you need to send multiple messages, the email backend API *provides an alternative*.

EmailMessage Objects

class `EmailMessage`

The `EmailMessage` class is initialized with the following parameters (in the given order, if positional arguments are used). All parameters are optional and can be set at any time prior to calling the `send()` method. Changed in version 1.3: The `cc` argument was added.

- `subject`: The subject line of the email.
- `body`: The body text. This should be a plain text message.
- `from_email`: The sender's address. Both `fred@example.com` and `Fred <fred@example.com>` forms are legal. If omitted, the `DEFAULT_FROM_EMAIL` setting is used.

- `to`: A list or tuple of recipient addresses.
- `bcc`: A list or tuple of addresses used in the "Bcc" header when sending the email.
- `connection`: An email backend instance. Use this parameter if you want to use the same connection for multiple messages. If omitted, a new connection is created when `send()` is called.
- `attachments`: A list of attachments to put on the message. These can be either `email.MIMEBase.MIMEBase` instances, or `(filename, content, mimetype)` triples.
- `headers`: A dictionary of extra headers to put on the message. The keys are the header name, values are the header values. It's up to the caller to ensure header names and values are in the correct format for an email message.
- `cc`: A list or tuple of recipient addresses used in the "Cc" header when sending the email.

For example:

```
email = EmailMessage('Hello', 'Body goes here', 'from@example.com',
            ['to1@example.com', 'to2@example.com'], ['bcc@example.com'],
            headers = {'Reply-To': 'another@example.com'})
```

The class has the following methods:

- `send(fail_silently=False)` sends the message. If a connection was specified when the email was constructed, that connection will be used. Otherwise, an instance of the default backend will be instantiated and used. If the keyword argument `fail_silently` is `True`, exceptions raised while sending the message will be quashed.
- `message()` constructs a `django.core.mail.SafeMIMEText` object (a subclass of Python's `email.MIMEText.MIMEText` class) or a `django.core.mail.SafeMIMEMultipart` object holding the message to be sent. If you ever need to extend the `EmailMessage` class, you'll probably want to override this method to put the content you want into the MIME object.
- `recipients()` returns a list of all the recipients of the message, whether they're recorded in the `to`, `cc` or `bcc` attributes. This is another method you might need to override when subclassing, because the SMTP server needs to be told the full list of recipients when the message is sent. If you add another way to specify recipients in your class, they need to be returned from this method as well.
- `attach()` creates a new file attachment and adds it to the message. There are two ways to call `attach()`:
  - You can pass it a single argument that is an `email.MIMEBase.MIMEBase` instance. This will be inserted directly into the resulting message.
  - Alternatively, you can pass `attach()` three arguments: `filename`, `content` and `mimetype`. `filename` is the name of the file attachment as it will appear in the email, `content` is the data that will be contained inside the attachment and `mimetype` is the optional MIME type for the attachment. If you omit `mimetype`, the MIME content type will be guessed from the filename of the attachment.

    For example:

    ```
    message.attach('design.png', img_data, 'image/png')
    ```

- `attach_file()` creates a new attachment using a file from your filesystem. Call it with the path of the file to attach and, optionally, the MIME type to use for the attachment. If the MIME type is omitted, it will be guessed from the filename. The simplest use would be:

  ```
  message.attach_file('/images/weather_map.png')
  ```

Sending alternative content types

It can be useful to include multiple versions of the content in an email; the classic example is to send both text and HTML versions of a message. With Django's email library, you can do this using the `EmailMultiAlternatives` class. This subclass of `EmailMessage` has an `attach_alternative()` method for including extra versions of the message body in the email. All the other methods (including the class initialization) are inherited directly from `EmailMessage`.

To send a text and HTML combination, you could write:

```
from django.core.mail import EmailMultiAlternatives

subject, from_email, to = 'hello', 'from@example.com', 'to@example.com'
text_content = 'This is an important message.'
html_content = '<p>This is an <strong>important</strong> message.</p>'
msg = EmailMultiAlternatives(subject, text_content, from_email, [to])
msg.attach_alternative(html_content, "text/html")
msg.send()
```

By default, the MIME type of the `body` parameter in an `EmailMessage` is `"text/plain"`. It is good practice to leave this alone, because it guarantees that any recipient will be able to read the email, regardless of their mail client. However, if you are confident that your recipients can handle an alternative content type, you can use the `content_subtype` attribute on the `EmailMessage` class to change the main content type. The major type will always be `"text"`, but you can change the subtype. For example:

```
msg = EmailMessage(subject, html_content, from_email, [to])
msg.content_subtype = "html"  # Main content is now text/html
msg.send()
```

3.13.9 Email backends

The actual sending of an email is handled by the email backend.

The email backend class has the following methods:

- `open()` instantiates an long-lived email-sending connection.
- `close()` closes the current email-sending connection.
- `send_messages(email_messages)` sends a list of `EmailMessage` objects. If the connection is not open, this call will implicitly open the connection, and close the connection afterwards. If the connection is already open, it will be left open after mail has been sent.

Obtaining an instance of an email backend

The `get_connection()` function in `django.core.mail` returns an instance of the email backend that you can use.

`get_connection`(*backend=None*, *fail_silently=False*, **args*, ***kwargs*)

By default, a call to `get_connection()` will return an instance of the email backend specified in `EMAIL_BACKEND`. If you specify the `backend` argument, an instance of that backend will be instantiated.

The `fail_silently` argument controls how the backend should handle errors. If `fail_silently` is True, exceptions during the email sending process will be silently ignored.

All other arguments are passed directly to the constructor of the email backend.

Django ships with several email sending backends. With the exception of the SMTP backend (which is the default), these backends are only useful during testing and development. If you have special email sending requirements, you can *write your own email backend*.

SMTP backend

This is the default backend. Email will be sent through a SMTP server. The server address and authentication credentials are set in the `EMAIL_HOST`, `EMAIL_PORT`, `EMAIL_HOST_USER`, `EMAIL_HOST_PASSWORD` and `EMAIL_USE_TLS` settings in your settings file.

The SMTP backend is the default configuration inherited by Django. If you want to specify it explicitly, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

Console backend

Instead of sending out real emails the console backend just writes the emails that would be send to the standard output. By default, the console backend writes to `stdout`. You can use a different stream-like object by providing the `stream` keyword argument when constructing the connection.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

File backend

The file backend writes emails to a file. A new file is created for each new session that is opened on this backend. The directory to which the files are written is either taken from the `EMAIL_FILE_PATH` setting or from the `file_path` keyword when creating a connection with `get_connection()`.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.filebased.EmailBackend'
EMAIL_FILE_PATH = '/tmp/app-messages' # change this to a proper location
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

In-memory backend

The `'locmem'` backend stores messages in a special attribute of the `django.core.mail` module. The `outbox` attribute is created when the first message is sent. It's a list with an `EmailMessage` instance for each message that would be send.

To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.locmem.EmailBackend'
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development and testing.

Dummy backend

As the name suggests the dummy backend does nothing with your messages. To specify this backend, put the following in your settings:

```
EMAIL_BACKEND = 'django.core.mail.backends.dummy.EmailBackend'
```

This backend is not intended for use in production – it is provided as a convenience that can be used during development.

Defining a custom email backend

If you need to change how emails are sent you can write your own email backend. The `EMAIL_BACKEND` setting in your settings file is then the Python import path for your backend class.

Custom email backends should subclass `BaseEmailBackend` that is located in the `django.core.mail.backends.base` module. A custom email backend must implement the `send_messages(email_messages)` method. This method receives a list of `EmailMessage` instances and returns the number of successfully delivered messages. If your backend has any concept of a persistent session or connection, you should also implement the `open()` and `close()` methods. Refer to `smtp.EmailBackend` for a reference implementation.

Sending multiple emails

Establishing and closing an SMTP connection (or any other network connection, for that matter) is an expensive process. If you have a lot of emails to send, it makes sense to reuse an SMTP connection, rather than creating and destroying a connection every time you want to send an email.

There are two ways you tell an email backend to reuse a connection.

Firstly, you can use the `send_messages()` method. `send_messages()` takes a list of `EmailMessage` instances (or subclasses), and sends them all using a single connection.

For example, if you have a function called `get_notification_email()` that returns a list of `EmailMessage` objects representing some periodic email you wish to send out, you could send these emails using a single call to send_messages:

```
from django.core import mail
connection = mail.get_connection()   # Use default email connection
messages = get_notification_email()
connection.send_messages(messages)
```

In this example, the call to `send_messages()` opens a connection on the backend, sends the list of messages, and then closes the connection again.

The second approach is to use the `open()` and `close()` methods on the email backend to manually control the connection. `send_messages()` will not manually open or close the connection if it is already open, so if you manually open the connection, you can control when it is closed. For example:

```
from django.core import mail
connection = mail.get_connection()

# Manually open the connection
connection.open()

# Construct an email message that uses the connection
email1 = mail.EmailMessage('Hello', 'Body goes here', 'from@example.com',
```

```
                          ['to1@example.com'], connection=connection)
email1.send() # Send the email

# Construct two more messages
email2 = mail.EmailMessage('Hello', 'Body goes here', 'from@example.com',
                          ['to2@example.com'])
email3 = mail.EmailMessage('Hello', 'Body goes here', 'from@example.com',
                          ['to3@example.com'])

# Send the two emails in a single call -
connection.send_messages([email2, email3])
# The connection was already open so send_messages() doesn't close it.
# We need to manually close the connection.
connection.close()
```

3.13.10 Testing email sending

There are times when you do not want Django to send emails at all. For example, while developing a Web site, you probably don't want to send out thousands of emails – but you may want to validate that emails will be sent to the right people under the right conditions, and that those emails will contain the correct content.

The easiest way to test your project's use of email is to use the `console` email backend. This backend redirects all email to stdout, allowing you to inspect the content of mail.

The `file` email backend can also be useful during development – this backend dumps the contents of every SMTP connection to a file that can be inspected at your leisure.

Another approach is to use a "dumb" SMTP server that receives the emails locally and displays them to the terminal, but does not actually send anything. Python has a built-in way to accomplish this with a single command:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

This command will start a simple SMTP server listening on port 1025 of localhost. This server simply prints to standard output all email headers and the email body. You then only need to set the `EMAIL_HOST` and `EMAIL_PORT` accordingly, and you are set.

For a more detailed discussion of testing and processing of emails locally, see the Python documentation for the `smtpd` module.

3.14 Internationalization and localization

3.14.1 Translation

Overview

In order to make a Django project translatable, you have to add a minimal amount of hooks to your Python code and templates. These hooks are called *translation strings*. They tell Django: "This text should be translated into the end user's language, if a translation for this text is available in that language." It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

Django then provides utilities to extract the translation strings into a *message file*. This file is a convenient way for translators to provide the equivalent of the translation strings in the target language. Once the translators have filled in the message file, it must be compiled. This process relies on the GNU gettext toolset.

Once this is done, Django takes care of translating Web apps on the fly in each available language, according to users' language preferences.

Django's internationalization hooks are on by default, and that means there's a bit of i18n-related overhead in certain places of the framework. If you don't use internationalization, you should take the two seconds to set `USE_I18N = False` in your settings file. Then Django will make some optimizations so as not to load the internationalization machinery. You'll probably also want to remove `'django.core.context_processors.i18n'` from your `TEMPLATE_CONTEXT_PROCESSORS` setting.

Note: There is also an independent but related `USE_L10N` setting that controls if Django should implement format localization. See *Format localization* for more details.

Internationalization: in Python code

Standard translation

Specify a translation string by using the function `ugettext()`. It's convention to import this as a shorter alias, _, to save typing.

Note: Python's standard library `gettext` module installs `_()` into the global namespace, as an alias for `gettext()`. In Django, we have chosen not to follow this practice, for a couple of reasons:

1. For international character set (Unicode) support, `ugettext()` is more useful than `gettext()`. Sometimes, you should be using `ugettext_lazy()` as the default translation method for a particular file. Without `_()` in the global namespace, the developer has to think about which is the most appropriate translation function.
2. The underscore character (_) is used to represent "the previous result" in Python's interactive shell and doctest tests. Installing a global `_()` function causes interference. Explicitly importing `ugettext()` as `_()` avoids this problem.

In this example, the text `"Welcome to my site."` is marked as a translation string:

```
from django.utils.translation import ugettext as _

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

Obviously, you could code this without using the alias. This example is identical to the previous one:

```
from django.utils.translation import ugettext

def my_view(request):
    output = ugettext("Welcome to my site.")
    return HttpResponse(output)
```

Translation works on computed values. This example is identical to the previous two:

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

Translation works on variables. Again, here's an identical example:

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

(The caveat with using variables or computed values, as in the previous two examples, is that Django's translation-string-detecting utility, `django-admin.py makemessages`, won't be able to find these strings. More on `makemessages` later.)

The strings you pass to `_()` or `ugettext()` can take placeholders, specified with Python's standard named-string interpolation syntax. Example:

```
def my_view(request, m, d):
    output = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
    return HttpResponse(output)
```

This technique lets language-specific translations reorder the placeholder text. For example, an English translation may be `"Today is November 26."`, while a Spanish translation may be `"Hoy es 26 de Noviembre."` – with the the month and the day placeholders swapped.

For this reason, you should use named-string interpolation (e.g., `%(day)s`) instead of positional interpolation (e.g., `%s` or `%d`) whenever you have more than a single parameter. If you used positional interpolation, translations wouldn't be able to reorder placeholder text.

Comments for translators

New in version 1.3: *Please see the release notes* If you would like to give translators hints about a translatable string, you can add a comment prefixed with the `Translators` keyword on the line preceding the string, e.g.:

```
def my_view(request):
    # Translators: This message appears on the home page only
    output = ugettext("Welcome to my site.")
```

This also works in templates with the `comment` tag:

```
{% comment %}Translators: This is a text of the base template {% endcomment %}
```

The comment will then appear in the resulting `.po` file and should also be displayed by most translation tools.

Marking strings as no-op

Use the function `django.utils.translation.ugettext_noop()` to mark a string as a translation string without translating it. The string is later translated from a variable.

Use this if you have constant strings that should be stored in the source language because they are exchanged over systems or users – such as strings in a database – but should be translated at the last possible point in time, such as when the string is presented to the user.

Pluralization

Use the function `django.utils.translation.ungettext()` to specify pluralized messages.

`ungettext` takes three arguments: the singular translation string, the plural translation string and the number of objects.

This function is useful when you need your Django application to be localizable to languages where the number and complexity of plural forms is greater than the two forms used in English ('object' for the singular and 'objects' for all the cases where `count` is different from one, irrespective of its value.)

For example:

```
from django.utils.translation import ungettext

def hello_world(request, count):
    page = ungettext(
        'there is %(count)d object',
        'there are %(count)d objects',
    count) % {
        'count': count,
    }
    return HttpResponse(page)
```

In this example the number of objects is passed to the translation languages as the `count` variable.

Lets see a slightly more complex usage example:

```
from django.utils.translation import ungettext

count = Report.objects.count()
if count == 1:
    name = Report._meta.verbose_name
else:
    name = Report._meta.verbose_name_plural

text = ungettext(
        'There is %(count)d %(name)s available.',
        'There are %(count)d %(name)s available.',
        count
) % {
    'count': count,
    'name': name
}
```

Here we reuse localizable, hopefully already translated literals (contained in the `verbose_name` and `verbose_name_plural` model `Meta` options) for other parts of the sentence so all of it is consistently based on the cardinality of the elements at play.

Note: When using this technique, make sure you use a single name for every extrapolated variable included in the literal. In the example above note how we used the `name` Python variable in both translation strings. This example would fail:

```
from django.utils.translation import ungettext
from myapp.models import Report

count = Report.objects.count()
d = {
    'count': count,
    'name': Report._meta.verbose_name,
    'plural_name': Report._meta.verbose_name_plural
}
text = ungettext(
        'There is %(count)d %(name)s available.',
        'There are %(count)d %(plural_name)s available.',
        count
```

```
) % d
```

You would get an error when running `django-admin.py compilemessages`:

```
a format specification for argument 'name', as in 'msgstr[0]', doesn't exist in 'msgid'
```

Contextual markers

New in version 1.3: *Please see the release notes* Sometimes words have several meanings, such as "May" in English, which refers to a month name and to a verb. To enable translators to translate these words correctly in different contexts, you can use the `django.utils.translation.pgettext()` function, or the `django.utils.translation.npgettext()` function if the string needs pluralization. Both take a context string as the first variable.

In the resulting `.po` file, the string will then appear as often as there are different contextual markers for the same string (the context will appear on the `msgctxt` line), allowing the translator to give a different translation for each of them.

For example:

```
from django.utils.translation import pgettext

month = pgettext("month name", "May")
```

or:

```
from django.utils.translation import pgettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=pgettext_lazy(
        'help text for MyThing model', 'This is the help text'))
```

will appear in the `.po` file as:

```
msgctxt "month name"
msgid "May"
msgstr ""
```

New in version 1.4: *Please see the release notes* Contextual markers are also supported by the `trans` and `blocktrans` template tags.

Lazy translation

Use the lazy versions of translation functions in `django.utils.translation` (easily recognizable by the `lazy` suffix in their names) to translate strings lazily – when the value is accessed rather than when they're called.

These functions store a lazy reference to the string – not the actual translation. The translation itself will be done when the string is used in a string context, such as in template rendering.

This is essential when calls to these functions are located in code paths that are executed at module load time.

This is something that can easily happen when defining models, forms and model forms, because Django implements these such that their fields are actually class-level attributes. For that reason, make sure to use lazy translations in the following cases:

Model fields and relationships `verbose_name` and `help_text` option values For example, to translate the help text of the *name* field in the following model, do the following:

```
from django.utils.translation import ugettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=ugettext_lazy('This is the help text'))
```

You can mark names of `ForeignKey`, `ManyTomanyField` or `OneToOneField` relationship as translatable by using their `verbose_name` options:

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind, related_name='kinds',
                             verbose_name=_('kind'))
```

Just like you would do in `verbose_name` you should provide a lowercase verbose name text for the relation as Django will automatically titlecase it when required.

Model verbose names values It is recommended to always provide explicit `verbose_name` and `verbose_name_plural` options rather than relying on the fallback English-centric and somewhat naïve determination of verbose names Django performs bu looking at the model's class name:

```
from django.utils.translation import ugettext_lazy

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=ugettext_lazy('This is the help text'))

    class Meta:
        verbose_name = ugettext_lazy('my thing')
        verbose_name_plural = ugettext_lazy('my things')
```

Model methods `short_description` attribute values For model methods, you can provide translations to Django and the admin site with the `short_description` attribute:

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    kind = models.ForeignKey(ThingKind, related_name='kinds',
                             verbose_name=_('kind'))

    def is_mouse(self):
        return self.kind.type == MOUSE_TYPE
    is_mouse.short_description = _('Is it a mouse?')
```

Working with lazy translation objects

The result of a `ugettext_lazy()` call can be used wherever you would use a unicode string (an object with type `unicode`) in Python. If you try to use it where a bytestring (a `str` object) is expected, things will not work as expected, since a `ugettext_lazy()` object doesn't know how to convert itself to a bytestring. You can't use a unicode string inside a bytestring, either, so this is consistent with normal Python behavior. For example:

```
# This is fine: putting a unicode proxy into a unicode string.
u"Hello %s" % ugettext_lazy("people")
```

```
# This will not work, since you cannot insert a unicode object
# into a bytestring (nor can you insert our unicode proxy there)
"Hello %s" % ugettext_lazy("people")
```

If you ever see output that looks like "hello <django.utils.functional...>", you have tried to insert the result of ugettext_lazy() into a bytestring. That's a bug in your code.

If you don't like the long ugettext_lazy name, you can just alias it as _ (underscore), like so:

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

Using ugettext_lazy() and ungettext_lazy() to mark strings in models and utility functions is a common operation. When you're working with these objects elsewhere in your code, you should ensure that you don't accidentally convert them to strings, because they should be converted as late as possible (so that the correct locale is in effect). This necessitates the use of the helper function described next.

Joining strings: string_concat() Standard Python string joins (''.join([...])) will not work on lists containing lazy translation objects. Instead, you can use django.utils.translation.string_concat(), which creates a lazy object that concatenates its contents *and* converts them to strings only when the result is included in a string. For example:

```
from django.utils.translation import string_concat
...
name = ugettext_lazy('John Lennon')
instrument = ugettext_lazy('guitar')
result = string_concat(name, ': ', instrument)
```

In this case, the lazy translations in result will only be converted to strings when result itself is used in a string (usually at template rendering time).

Localized names of languages

get_language_info()

New in version 1.3: *Please see the release notes* The get_language_info() function provides detailed information about languages:

```
>>> from django.utils.translation import get_language_info
>>> li = get_language_info('de')
>>> print(li['name'], li['name_local'], li['bidi'])
German Deutsch False
```

The name and name_local attributes of the dictionary contain the name of the language in English and in the language itself, respectively. The bidi attribute is True only for bi-directional languages.

The source of the language information is the django.conf.locale module. Similar access to this information is available for template code. See below.

Internationalization: in template code

Translations in *Django templates* uses two template tags and a slightly different syntax than in Python code. To give your template access to these tags, put {% load i18n %} toward the top of your template.

`trans` template tag

The `{% trans %}` template tag translates either a constant string (enclosed in single or double quotes) or variable content:

```
<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>
```

If the `noop` option is present, variable lookup still takes place but the translation is skipped. This is useful when "stubbing out" content that will require translation in the future:

```
<title>{% trans "myvar" noop %}</title>
```

Internally, inline translations use an `ugettext()` call.

In case a template var (`myvar` above) is passed to the tag, the tag will first resolve such variable to a string at run-time and then look up that string in the message catalogs.

It's not possible to mix a template variable inside a string within `{% trans %}`. If your translations require strings with variables (placeholders), use `{% blocktrans %}` instead. New in version 1.4: *Please see the release notes* If you'd like to retrieve a translated string without displaying it, you can use the following syntax:

```
{% trans "This is the title" as the_title %}

<title>{{ the_title }}</title>
<meta name="description" content="{{ the_title }}">
```

In practice you'll use this to get strings that are used in multiple places or should be used as arguments for other template tags or filters:

```
{% trans "starting point" as start %}
{% trans "end point" as end %}
{% trans "La Grande Boucle" as race %}

<h1>
  <a href="/" title="{% blocktrans %}Back to '{{ race }}' homepage{% endblocktrans %}">{{ race }}</a
</h1>
<p>
{% for stage in tour_stages %}
    {% cycle start end %}: {{ stage }}{% if forloop.counter|divisibleby:2 %}<br />{% else %}, {% end
{% endfor %}
</p>
```

New in version 1.4: *Please see the release notes* `{% trans %}` also supports *contextual markers* using the `context` keyword:

```
{% trans "May" context "month name" %}
```

`blocktrans` template tag

Changed in version 1.3: New keyword argument format. Contrarily to the `trans` tag, the `blocktrans` tag allows you to mark complex sentences consisting of literals and variable content for translation by making use of placeholders:

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

To translate a template expression – say, accessing object attributes or using template filters – you need to bind the expression to a local variable for use within the translation block. Examples:

```
{% blocktrans with amount=article.price %}
That will cost $ {{ amount }}.
{% endblocktrans %}

{% blocktrans with myvar=value|filter %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

You can use multiple expressions inside a single `blocktrans` tag:

```
{% blocktrans with book_t=book|title author_t=author|title %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

Note: The previous more verbose format is still supported: `{% blocktrans with book|title as book_t and author|title as author_t %}`

Changed in version 1.4: *Please see the release notes* If resolving one of the block arguments fails, blocktrans will fall back to the default language by deactivating the currently active language temporarily with the `deactivate_all()` function.

This tag also provides for pluralization. To use it:

- Designate and bind a counter value with the name `count`. This value will be the one used to select the right plural form.
- Specify both the singular and plural forms separating them with the `{% plural %}` tag within the `{% blocktrans %}` and `{% endblocktrans %}` tags.

An example:

```
{% blocktrans count counter=list|length %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

A more complex example:

```
{% blocktrans with amount=article.price count years=i.length %}
That will cost $ {{ amount }} per year.
{% plural %}
That will cost $ {{ amount }} per {{ years }} years.
{% endblocktrans %}
```

When you use both the pluralization feature and bind values to local variables in addition to the counter value, keep in mind that the `blocktrans` construct is internally converted to an `ungettext` call. This means the same *notes regarding ungettext variables* apply.

Reverse URL lookups cannot be carried out within the `blocktrans` and should be retrieved (and stored) beforehand:

```
{% url 'path.to.view' arg arg2 as the_url %}
{% blocktrans %}
This is a URL: {{ the_url }}
{% endblocktrans %}
```

New in version 1.4: *Please see the release notes* `{% blocktrans %}` also supports *contextual markers* using the `context` keyword:

```
{% blocktrans with name=user.username context "greeting" %}Hi {{ name }}{% endblocktrans %}
```

Other tags

Each `RequestContext` has access to three translation-specific variables:

- `LANGUAGES` is a list of tuples in which the first element is the *language code* and the second is the language name (translated into the currently active locale).
- `LANGUAGE_CODE` is the current user's preferred language, as a string. Example: `en-us`. (See *How Django discovers language preference*.)
- `LANGUAGE_BIDI` is the current locale's direction. If True, it's a right-to-left language, e.g.: Hebrew, Arabic. If False it's a left-to-right language, e.g.: English, French, German etc.

If you don't use the `RequestContext` extension, you can get those values with three tags:

```
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_current_language_bidi as LANGUAGE_BIDI %}
```

These tags also require a `{% load i18n %}`.

Translation hooks are also available within any template block tag that accepts constant strings. In those cases, just use `_()` syntax to specify a translation string:

```
{% some_special_tag _("Page not found") value|yesno:_("yes,no") %}
```

In this case, both the tag and the filter will see the already-translated string, so they don't need to be aware of translations.

Note: In this example, the translation infrastructure will be passed the string `"yes,no"`, not the individual strings `"yes"` and `"no"`. The translated string will need to contain the comma so that the filter parsing code knows how to split up the arguments. For example, a German translator might translate the string `"yes,no"` as `"ja,nein"` (keeping the comma intact).

New in version 1.3: *Please see the release notes* You can also retrieve information about any of the available languages using provided template tags and filters. To get information about a single language, use the `{% get_language_info %}` tag:

```
{% get_language_info for LANGUAGE_CODE as lang %}
{% get_language_info for "pl" as lang %}
```

You can then access the information:

```
Language code: {{ lang.code }}<br />
Name of language: {{ lang.name_local }}<br />
Name in English: {{ lang.name }}<br />
Bi-directional: {{ lang.bidi }}
```

You can also use the `{% get_language_info_list %}` template tag to retrieve information for a list of languages (e.g. active languages as specified in `LANGUAGES`). See *the section about the set_language redirect view* for an example of how to display a language selector using `{% get_language_info_list %}`.

In addition to `LANGUAGES` style nested tuples, `{% get_language_info_list %}` supports simple lists of language codes. If you do this in your view:

```
return render_to_response('mytemplate.html', {
    'available_languages': ['en', 'es', 'fr'],
}, RequestContext(request))
```

you can iterate over those languages in the template:

```
{% get_language_info_list for available_languages as langs %}
{% for lang in langs %} ... {% endfor %}
```

There are also simple filters available for convenience:

- `{{ LANGUAGE_CODE|language_name }}` ("German")
- `{{ LANGUAGE_CODE|language_name_local }}` ("Deutsch")
- `{{ LANGUAGE_CODE|bidi }}` (False)

Internationalization: in JavaScript code

Adding translations to JavaScript poses some problems:

- JavaScript code doesn't have access to a `gettext` implementation.
- JavaScript code doesn't have access to `.po` or `.mo` files; they need to be delivered by the server.
- The translation catalogs for JavaScript should be kept as small as possible.

Django provides an integrated solution for these problems: It passes the translations into JavaScript, so you can call `gettext`, etc., from within JavaScript.

The `javascript_catalog` view

`javascript_catalog`(*request*, *domain='djangojs'*, *packages=None*)

The main solution to these problems is the `django.views.i18n.javascript_catalog()` view, which sends out a JavaScript code library with functions that mimic the `gettext` interface, plus an array of translation strings. Those translation strings are taken from applications or Django core, according to what you specify in either the `info_dict` or the URL. Paths listed in `LOCALE_PATHS` are also included.

You hook it up like this:

```
js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = patterns('',
    (r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_dict),
)
```

Each string in `packages` should be in Python dotted-package syntax (the same format as the strings in `INSTALLED_APPS`) and should refer to a package that contains a `locale` directory. If you specify multiple packages, all those catalogs are merged into one catalog. This is useful if you have JavaScript that uses strings from different applications.

The precedence of translations is such that the packages appearing later in the `packages` argument have higher precedence than the ones appearing at the beginning, this is important in the case of clashing translations for the same literal.

By default, the view uses the `djangojs` gettext domain. This can be changed by altering the `domain` argument.

You can make the view dynamic by putting the packages into the URL pattern:

```
urlpatterns = patterns('',
    (r'^jsi18n/(?P<packages>\S+?)/$', 'django.views.i18n.javascript_catalog'),
)
```

With this, you specify the packages as a list of package names delimited by '+' signs in the URL. This is especially useful if your pages use code from different apps and this changes often and you don't want to pull in one big catalog file. As a security measure, these values can only be either `django.conf` or any package from the `INSTALLED_APPS` setting.

The JavaScript translations found in the paths listed in the `LOCALE_PATHS` setting are also always included. To keep consistency with the translations lookup order algorithm used for Python and templates, the directories listed in `LOCALE_PATHS` have the highest precedence with the ones appearing first having higher precedence than the ones appearing later. Changed in version 1.3: Directories listed in `LOCALE_PATHS` weren't included in the lookup algorithm until version 1.3.

Using the JavaScript translation catalog

To use the catalog, just pull in the dynamically generated script like this:

```
<script type="text/javascript" src="{% url 'django.views.i18n.javascript_catalog' %}"></script>
```

This uses reverse URL lookup to find the URL of the JavaScript catalog view. When the catalog is loaded, your JavaScript code can use the standard `gettext` interface to access it:

```
document.write(gettext('this is to be translated'));
```

There is also an `ngettext` interface:

```
var object_cnt = 1 // or 0, or 2, or 3, ...
s = ngettext('literal for the singular case',
        'literal for the plural case', object_cnt);
```

and even a string interpolation function:

```
function interpolate(fmt, obj, named);
```

The interpolation syntax is borrowed from Python, so the `interpolate` function supports both positional and named interpolation:

- Positional interpolation: `obj` contains a JavaScript Array object whose elements values are then sequentially interpolated in their corresponding `fmt` placeholders in the same order they appear. For example:

  ```
  fmts = ngettext('There is %s object. Remaining: %s',
          'There are %s objects. Remaining: %s', 11);
  s = interpolate(fmts, [11, 20]);
  // s is 'There are 11 objects. Remaining: 20'
  ```

- Named interpolation: This mode is selected by passing the optional boolean `named` parameter as true. `obj` contains a JavaScript object or associative array. For example:

  ```
  d = {
      count: 10,
      total: 50
  };

  fmts = ngettext('Total: %(total)s, there is %(count)s object',
  ```

```
    'there are %(count)s of a total of %(total)s objects', d.count);
    s = interpolate(fmts, d, true);
```

You shouldn't go over the top with string interpolation, though: this is still JavaScript, so the code has to make repeated regular-expression substitutions. This isn't as fast as string interpolation in Python, so keep it to those cases where you really need it (for example, in conjunction with `ngettext` to produce proper pluralizations).

Internationalization: in URL patterns

New in version 1.4: *Please see the release notes* Django provides two mechanisms to internationalize URL patterns:

- Adding the language prefix to the root of the URL patterns to make it possible for `LocaleMiddleware` to detect the language to activate from the requested URL.
- Making URL patterns themselves translatable via the `django.utils.translation.ugettext_lazy()` function.

Warning: Using either one of these features requires that an active language be set for each request; in other words, you need to have `django.middleware.locale.LocaleMiddleware` in your `MIDDLEWARE_CLASSES` setting.

Language prefix in URL patterns

i18n_patterns (*prefix*, *pattern_description*, ...)

This function can be used in your root URLconf as a replacement for the normal `django.conf.urls.patterns()` function. Django will automatically prepend the current active language code to all url patterns defined within `i18n_patterns()`. Example URL patterns:

```
from django.conf.urls import patterns, include, url
from django.conf.urls.i18n import i18n_patterns

urlpatterns = patterns('',
    url(r'^sitemap\.xml$', 'sitemap.view', name='sitemap_xml'),
)

news_patterns = patterns('',
    url(r'^$', 'news.views.index', name='index'),
    url(r'^category/(?P<slug>[\w-]+)/$', 'news.views.category', name='category'),
    url(r'^(?P<slug>[\w-]+)/$', 'news.views.details', name='detail'),
)

urlpatterns += i18n_patterns('',
    url(r'^about/$', 'about.view', name='about'),
    url(r'^news/', include(news_patterns, namespace='news')),
)
```

After defining these URL patterns, Django will automatically add the language prefix to the URL patterns that were added by the `i18n_patterns` function. Example:

```
from django.core.urlresolvers import reverse
from django.utils.translation import activate

>>> activate('en')
>>> reverse('sitemap_xml')
'/sitemap.xml'
```

```
>>> reverse('news:index')
'/en/news/'

>>> activate('nl')
>>> reverse('news:detail', kwargs={'slug': 'news-slug'})
'/nl/news/news-slug/'
```

Warning: `i18n_patterns()` is only allowed in your root URLconf. Using it within an included URLconf will throw an `ImproperlyConfigured` exception.

Warning: Ensure that you don't have non-prefixed URL patterns that might collide with an automatically-added language prefix.

Translating URL patterns

URL patterns can also be marked translatable using the `ugettext_lazy()` function. Example:

```
from django.conf.urls import patterns, include, url
from django.conf.urls.i18n import i18n_patterns
from django.utils.translation import ugettext_lazy as _

urlpatterns = patterns('',
    url(r'^sitemap\.xml$', 'sitemap.view', name='sitemap_xml'),
)

news_patterns = patterns('',
    url(r'^$', 'news.views.index', name='index'),
    url(_(r'^category/(?P<slug>[\w-]+)/$'), 'news.views.category', name='category'),
    url(r'^(?P<slug>[\w-]+)/$', 'news.views.details', name='detail'),
)

urlpatterns += i18n_patterns('',
    url(_(r'^about/$'), 'about.view', name='about'),
    url(_(r'^news/'), include(news_patterns, namespace='news')),
)
```

After you've created the translations, the `reverse()` function will return the URL in the active language. Example:

```
from django.core.urlresolvers import reverse
from django.utils.translation import activate

>>> activate('en')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/en/news/category/recent/'

>>> activate('nl')
>>> reverse('news:category', kwargs={'slug': 'recent'})
'/nl/nieuws/categorie/recent/'
```

Warning: In most cases, it's best to use translated URLs only within a language-code-prefixed block of patterns (using `i18n_patterns()`), to avoid the possibility that a carelessly translated URL causes a collision with a non-translated URL pattern.

Reversing in templates

If localized URLs get reversed in templates they always use the current language. To link to a URL in another language use the `language` template tag. It enables the given language in the enclosed template section:

```
{% load i18n %}

{% get_available_languages as languages %}

{% trans "View this category in:" %}
{% for lang_code, lang_name in languages %}
    {% language lang_code %}
    <a href="{% url 'category' slug=category.slug %}">{{ lang_name }}</a>
    {% endlanguage %}
{% endfor %}
```

The `language` tag expects the language code as the only argument.

Localization: how to create language files

Once the string literals of an application have been tagged for later translation, the translation themselves need to be written (or obtained). Here's how that works.

Locale restrictions

Django does not support localizing your application into a locale for which Django itself has not been translated. In this case, it will ignore your translation files. If you were to try this and Django supported it, you would inevitably see a mixture of translated strings (from your application) and English strings (from Django itself). If you want to support a locale for your application that is not already part of Django, you'll need to make at least a minimal translation of the Django core.

A good starting point is to copy the Django English `.po` file and to translate at least some *translation strings*.

Message files

The first step is to create a *message file* for a new language. A message file is a plain-text file, representing a single language, that contains all available translation strings and how they should be represented in the given language. Message files have a `.po` file extension.

Django comes with a tool, `django-admin.py makemessages`, that automates the creation and upkeep of these files.

Gettext utilities

The `makemessages` command (and `compilemessages` discussed later) use commands from the GNU gettext toolset: `xgettext`, `msgfmt`, `msgmerge` and `msguniq`.

The minimum version of the `gettext` utilities supported is 0.15.

To create or update a message file, run this command:

```
django-admin.py makemessages -l de
```

...where `de` is the language code for the message file you want to create. The language code, in this case, is in *locale format*. For example, it's `pt_BR` for Brazilian Portuguese and `de_AT` for Austrian German.

The script should be run from one of two places:

- The root directory of your Django project.
- The root directory of your Django app.

The script runs over your project source tree or your application source tree and pulls out all strings marked for translation. It creates (or updates) a message file in the directory `locale/LANG/LC_MESSAGES`. In the `de` example, the file will be `locale/de/LC_MESSAGES/django.po`.

By default `django-admin.py makemessages` examines every file that has the `.html` or `.txt` file extension. In case you want to override that default, use the `--extension` or `-e` option to specify the file extensions to examine:

```
django-admin.py makemessages -l de -e txt
```

Separate multiple extensions with commas and/or use `-e` or `--extension` multiple times:

```
django-admin.py makemessages -l de -e html,txt -e xml
```

Warning: When *creating message files from JavaScript source code* you need to use the special 'djangojs' domain, **not** `-e js`.

No gettext?

If you don't have the `gettext` utilities installed, `makemessages` will create empty files. If that's the case, either install the `gettext` utilities or just copy the English message file (`locale/en/LC_MESSAGES/django.po`) if available and use it as a starting point; it's just an empty translation file.

Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so `makemessages` works, see *gettext on Windows* for more information.

The format of `.po` files is straightforward. Each `.po` file contains a small bit of metadata, such as the translation maintainer's contact information, but the bulk of the file is a list of **messages** – simple mappings between translation strings and the actual translated text for the particular language.

For example, if your Django app contained a translation string for the text `"Welcome to my site."`, like so:

```
_("Welcome to my site.")
```

...then `django-admin.py makemessages` will have created a `.po` file containing the following snippet – a message:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

A quick explanation:

- `msgid` is the translation string, which appears in the source. Don't change it.
- `msgstr` is where you put the language-specific translation. It starts out empty, so it's your responsibility to change it. Make sure you keep the quotes around your translation.
- As a convenience, each message includes, in the form of a comment line prefixed with `#` and located above the `msgid` line, the filename and line number from which the translation string was gleaned.

Long messages are a special case. There, the first string directly after the `msgstr` (or `msgid`) is an empty string. Then the content itself will be written over the next few lines as one string per line. Those strings are directly concatenated. Don't forget trailing spaces within the strings; otherwise, they'll be tacked together without whitespace!

Mind your charset

When creating a PO file with your favorite text editor, first edit the charset line (search for `"CHARSET"`) and set it to the charset you'll be using to edit the content. Due to the way the `gettext` tools work internally and because we want to allow non-ASCII source strings in Django's core and your applications, you **must** use UTF-8 as the encoding for your PO file. This means that everybody will be using the same encoding, which is important when Django processes the PO files.

To reexamine all source code and templates for new translation strings and update all message files for **all** languages, run this:

```
django-admin.py makemessages -a
```

Compiling message files

After you create your message file – and each time you make changes to it – you'll need to compile it into a more efficient form, for use by `gettext`. Do this with the `django-admin.py compilemessages` utility.

This tool runs over all available `.po` files and creates `.mo` files, which are binary files optimized for use by `gettext`. In the same directory from which you ran `django-admin.py makemessages`, run `django-admin.py compilemessages` like this:

```
django-admin.py compilemessages
```

That's it. Your translations are ready for use.

Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so `django-admin.py compilemessages` works see *gettext on Windows* for more information.

.po files: Encoding and BOM usage.

Django only supports `.po` files encoded in UTF-8 and without any BOM (Byte Order Mark) so if your text editor adds such marks to the beginning of files by default then you will need to reconfigure it.

Creating message files from JavaScript source code

You create and update the message files the same way as the other Django message files – with the `django-admin.py makemessages` tool. The only difference is you need to explicitly specify what in gettext parlance is known as a domain in this case the `djangojs` domain, by providing a `-d djangojs` parameter, like this:

```
django-admin.py makemessages -d djangojs -l de
```

This would create or update the message file for JavaScript for German. After updating message files, just run `django-admin.py compilemessages` the same way as you do with normal Django message files.

gettext on Windows

This is only needed for people who either want to extract message IDs or compile message files (`.po`). Translation work itself just involves editing existing files of this type, but if you want to create your own message files, or want to test or compile a changed message file, you will need the `gettext` utilities:

- Download the following zip files from the GNOME servers http://ftp.gnome.org/pub/gnome/binaries/win32/dependencies/ or from one of its mirrors
  - `gettext-runtime-X.zip`
  - `gettext-tools-X.zip`

  `X` is the version number, we are requiring `0.15` or higher.
- Extract the contents of the `bin\` directories in both files to the same folder on your system (i.e. `C:\Program Files\gettext-utils`)
- Update the system PATH:
  - `Control Panel > System > Advanced > Environment Variables`.
  - In the `System variables` list, click `Path`, click `Edit`.
  - Add `;C:\Program Files\gettext-utils\bin` at the end of the `Variable value` field.

You may also use `gettext` binaries you have obtained elsewhere, so long as the `xgettext --version` command works properly. Do not attempt to use Django translation utilities with a `gettext` package if the command `xgettext --version` entered at a Windows command prompt causes a popup window saying "xgettext.exe has generated errors and will be closed by Windows".

Miscellaneous

The set_language redirect view

set_language(*request*)

As a convenience, Django comes with a view, `django.views.i18n.set_language()`, that sets a user's language preference and redirects to a given URL or, by default, back to the previous page.

Activate this view by adding the following line to your URLconf:

```
(r'^i18n/', include('django.conf.urls.i18n')),
```

(Note that this example makes the view available at `/i18n/setlang/`.)

The view expects to be called via the `POST` method, with a `language` parameter set in request. If session support is enabled, the view saves the language choice in the user's session. Otherwise, it saves the language choice in a cookie that is by default named `django_language`. (The name can be changed through the `LANGUAGE_COOKIE_NAME` setting.)

After setting the language choice, Django redirects the user, following this algorithm:

- Django looks for a `next` parameter in the `POST` data.
- If that doesn't exist, or is empty, Django tries the URL in the `Referrer` header.
- If that's empty – say, if a user's browser suppresses that header – then the user will be redirected to `/` (the site root) as a fallback.

Here's example HTML template code:

```
<form action="/i18n/setlang/" method="post">
{% csrf_token %}
<input name="next" type="hidden" value="{{ redirect_to }}" />
<select name="language">
{% get_language_info_list for LANGUAGES as languages %}
{% for language in languages %}
<option value="{{ language.code }}">{{ language.name_local }} ({{ language.code }})</option>
{% endfor %}
</select>
<input type="submit" value="Go" />
</form>
```

In this example, Django looks up the URL of the page to which the user will be redirected in the `redirect_to` context variable.

Using translations outside views and templates

While Django provides a rich set of i18n tools for use in views and templates, it does not restrict the usage to Django-specific code. The Django translation mechanisms can be used to translate arbitrary texts to any language that is supported by Django (as long as an appropriate translation catalog exists, of course). You can load a translation catalog, activate it and translate text to language of your choice, but remember to switch back to original language, as activating a translation catalog is done on per-thread basis and such change will affect code running in the same thread.

For example:

```
from django.utils import translation
def welcome_translated(language):
    cur_language = translation.get_language()
    try:
        translation.activate(language)
        text = translation.ugettext('welcome')
    finally:
        translation.activate(cur_language)
    return text
```

Calling this function with the value 'de' will give you `"Willkommen"`, regardless of `LANGUAGE_CODE` and language set by middleware.

Functions of particular interest are `django.utils.translation.get_language()` which returns the language used in the current thread, `django.utils.translation.activate()` which activates a translation catalog for the current thread, and `django.utils.translation.check_for_language()` which checks if the given language is supported by Django.

Implementation notes

Specialties of Django translation

Django's translation machinery uses the standard `gettext` module that comes with Python. If you know `gettext`, you might note these specialties in the way Django does translation:

- The string domain is `django` or `djangojs`. This string domain is used to differentiate between different programs that store their data in a common message-file library (usually `/usr/share/locale/`). The `django` domain is used for python and template translation strings and is loaded into the global translation catalogs. The `djangojs` domain is only used for JavaScript translation catalogs to make sure that those are as small as possible.

- Django doesn't use `xgettext` alone. It uses Python wrappers around `xgettext` and `msgfmt`. This is mostly for convenience.

How Django discovers language preference

Once you've prepared your translations – or, if you just want to use the translations that come with Django – you'll just need to activate translation for your app.

Behind the scenes, Django has a very flexible model of deciding which language should be used – installation-wide, for a particular user, or both.

To set an installation-wide language preference, set `LANGUAGE_CODE`. Django uses this language as the default translation – the final attempt if no other translator finds a translation.

If all you want to do is run Django with your native language, and a language file is available for it, all you need to do is set `LANGUAGE_CODE`.

If you want to let each individual user specify which language he or she prefers, use `LocaleMiddleware`. `LocaleMiddleware` enables language selection based on data from the request. It customizes content for each user.

To use `LocaleMiddleware`, add `'django.middleware.locale.LocaleMiddleware'` to your `MIDDLEWARE_CLASSES` setting. Because middleware order matters, you should follow these guidelines:

- Make sure it's one of the first middlewares installed.
- It should come after `SessionMiddleware`, because `LocaleMiddleware` makes use of session data. And it should come before `CommonMiddleware` because `CommonMiddleware` needs an activated language in order to resolve the requested URL.
- If you use `CacheMiddleware`, put `LocaleMiddleware` after it.

For example, your `MIDDLEWARE_CLASSES` might look like this:

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
)
```

(For more on middleware, see the *middleware documentation*.)

`LocaleMiddleware` tries to determine the user's language preference by following this algorithm: Changed in version 1.4: *Please see the release notes*

- First, it looks for the language prefix in the requested URL. This is only performed when you are using the `i18n_patterns` function in your root URLconf. See *Internationalization: in URL patterns* for more information about the language prefix and how to internationalize URL patterns.
- Failing that, it looks for a `django_language` key in the current user's session.
- Failing that, it looks for a cookie.

  The name of the cookie used is set by the `LANGUAGE_COOKIE_NAME` setting. (The default name is `django_language`.)
- Failing that, it looks at the `Accept-Language` HTTP header. This header is sent by your browser and tells the server which language(s) you prefer, in order by priority. Django tries each language in the header until it finds one with available translations.
- Failing that, it uses the global `LANGUAGE_CODE` setting.

Notes:

- In each of these places, the language preference is expected to be in the standard *language format*, as a string. For example, Brazilian Portuguese is `pt-br`.
- If a base language is available but the sublanguage specified is not, Django uses the base language. For example, if a user specifies `de-at` (Austrian German) but Django only has `de` available, Django uses `de`.
- Only languages listed in the `LANGUAGES` setting can be selected. If you want to restrict the language selection to a subset of provided languages (because your application doesn't provide all those languages), set `LANGUAGES` to a list of languages. For example:

  ```
  LANGUAGES = (
    ('de', _('German')),
    ('en', _('English')),
  )
  ```

  This example restricts languages that are available for automatic selection to German and English (and any sublanguage, like de-ch or en-us).
- If you define a custom `LANGUAGES` setting, as explained in the previous bullet, it's OK to mark the languages as translation strings – but use a "dummy" `ugettext()` function, not the one in `django.utils.translation`. You should *never* import `django.utils.translation` from within your settings file, because that module in itself depends on the settings, and that would cause a circular import.

  The solution is to use a "dummy" `ugettext()` function. Here's a sample settings file:

  ```
  ugettext = lambda s: s

  LANGUAGES = (
      ('de', ugettext('German')),
      ('en', ugettext('English')),
  )
  ```

  With this arrangement, `django-admin.py makemessages` will still find and mark these strings for translation, but the translation won't happen at runtime – so you'll have to remember to wrap the languages in the *real* `ugettext()` in any code that uses `LANGUAGES` at runtime.
- The `LocaleMiddleware` can only select languages for which there is a Django-provided base translation. If you want to provide translations for your application that aren't already in the set of translations in Django's source tree, you'll want to provide at least a basic one as described in the *Locale restrictions* note.

Once `LocaleMiddleware` determines the user's preference, it makes this preference available as `request.LANGUAGE_CODE` for each `HttpRequest`. Feel free to read this value in your view code. Here's a simple example:

```
def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

Note that, with static (middleware-less) translation, the language is in `settings.LANGUAGE_CODE`, while with dynamic (middleware) translation, it's in `request.LANGUAGE_CODE`.

How Django discovers translations

At runtime, Django builds an in-memory unified catalog of literals-translations. To achieve this it looks for translations by following this algorithm regarding the order in which it examines the different file paths to load the compiled *message files* (`.mo`) and the precedence of multiple translations for the same literal:

1. The directories listed in LOCALE_PATHS have the highest precedence, with the ones appearing first having higher precedence than the ones appearing later.
2. Then, it looks for and uses if it exists a locale directory in each of the installed apps listed in INSTALLED_APPS. The ones appearing first have higher precedence than the ones appearing later.
3. Finally, the Django-provided base translation in django/conf/locale is used as a fallback.

See Also:

The translations for literals included in JavaScript assets are looked up following a similar but not identical algorithm. See the *javascript_catalog view documentation* for more details.

In all cases the name of the directory containing the translation is expected to be named using *locale name* notation. E.g. de, pt_BR, es_AR, etc.

This way, you can write applications that include their own translations, and you can override base translations in your project. Or, you can just build a big project out of several apps and put all translations into one big common message file specific to the project you are composing. The choice is yours.

All message file repositories are structured the same way. They are:

- All paths listed in LOCALE_PATHS in your settings file are searched for <language>/LC_MESSAGES/django.(po|mo)
- $APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)
- $PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)

To create message files, you use the django-admin.py makemessages tool. You only need to be in the same directory where the locale/ directory is located. And you use django-admin.py compilemessages to produce the binary .mo files that are used by gettext.

You can also run django-admin.py compilemessages --settings=path.to.settings to make the compiler process all the directories in your LOCALE_PATHS setting.

Finally, you should give some thought to the structure of your translation files. If your applications need to be delivered to other users and will be used in other projects, you might want to use app-specific translations. But using app-specific translations and project-specific translations could produce weird problems with makemessages: it will traverse all directories below the current path and so might put message IDs into a unified, common message file for the current project that are already in application message files.

The easiest way out is to store applications that are not part of the project (and so carry their own translations) outside the project tree. That way, django-admin.py makemessages, when ran on a project level will only extract strings that are connected to your explicit project and not strings that are distributed independently.

3.14.2 Format localization

Overview

Django's formatting system is capable to display dates, times and numbers in templates using the format specified for the current *locale*. It also handles localized input in forms.

When it's enabled, two users accessing the same content may see dates, times and numbers formatted in different ways, depending on the formats for their current locale.

The formatting system is disabled by default. To enable it, it's necessary to set USE_L10N = True in your settings file.

Note: The default settings.py file created by django-admin.py startproject includes USE_L10N = True for convenience.

Note: There is also an independent but related `USE_I18N` setting that controls if Django should activate translation. See *Translation* for more details.

Locale aware input in forms

When formatting is enabled, Django can use localized formats when parsing dates, times and numbers in forms. That means it tries different formats for different locales when guessing the format used by the user when inputting data on forms.

Note: Django uses different formats for displaying data to those it uses for parsing data. Most notably, the formats for parsing dates can't use the `%a` (abbreviated weekday name), `%A` (full weekday name), `%b` (abbreviated month name), `%B` (full month name), or `%p` (AM/PM).

To enable a form field to localize input and output data simply use its `localize` argument:

```
class CashRegisterForm(forms.Form):
   product = forms.CharField()
   revenue = forms.DecimalField(max_digits=4, decimal_places=2, localize=True)
```

Controlling localization in templates

When you have enabled formatting with `USE_L10N`, Django will try to use a locale specific format whenever it outputs a value in a template.

However, it may not always be appropriate to use localized values – for example, if you're outputting Javascript or XML that is designed to be machine-readable, you will always want unlocalized values. You may also want to use localization in selected templates, rather than using localization everywhere.

To allow for fine control over the use of localization, Django provides the `l10n` template library that contains the following tags and filters.

Template tags

localize New in version 1.3: *Please see the release notes* Enables or disables localization of template variables in the contained block.

This tag allows a more fine grained control of localization than `USE_L10N`.

To activate or deactivate localization for a template block, use:

```
{% load l10n %}

{% localize on %}
    {{ value }}
{% endlocalize %}

{% localize off %}
    {{ value }}
{% endlocalize %}
```

Note: The value of `USE_L10N` isn't respected inside of a `{% localize %}` block.

See `localize` and `unlocalize` for template filters that will do the same job on a per-variable basis.

Template filters

localize New in version 1.3: *Please see the release notes* Forces localization of a single value.

For example:

```
{% load l10n %}

{{ value|localize }}
```

To disable localization on a single value, use `unlocalize`. To control localization over a large section of a template, use the `localize` template tag.

unlocalize New in version 1.3: *Please see the release notes* Forces a single value to be printed without localization.

For example:

```
{% load l10n %}

{{ value|unlocalize }}
```

To force localization of a single value, use `localize`. To control localization over a large section of a template, use the `localize` template tag.

Creating custom format files

Django provides format definitions for many locales, but sometimes you might want to create your own, because a format files doesn't exist for your locale, or because you want to overwrite some of the values.

To use custom formats, specify the path where you'll place format files first. To do that, just set your `FORMAT_MODULE_PATH` setting to the package where format files will exist, for instance:

```
FORMAT_MODULE_PATH = 'mysite.formats'
```

Files are not placed directly in this directory, but in a directory named as the locale, and must be named `formats.py`.

To customize the English formats, a structure like this would be needed:

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

where `formats.py` contains custom format definitions. For example:

```
THOUSAND_SEPARATOR = u'\xa0'
```

to use a non-breaking space (Unicode `00A0`) as a thousand separator, instead of the default for English, a comma.

Limitations of the provided locale formats

Some locales use context-sensitive formats for numbers, which Django's localization system cannot handle automatically.

Switzerland (German)

The Swiss number formatting depends on the type of number that is being formatted. For monetary values, a comma is used as the thousand separator and a decimal point for the decimal separator. For all other numbers, a comma is used as decimal separator and a space as thousand separator. The locale format provided by Django uses the generic separators, a comma for decimal and a space for thousand separators.

3.14.3 Time zones

New in version 1.4: *Please see the release notes*

Overview

When support for time zones is enabled, Django stores date and time information in UTC in the database, uses time-zone-aware datetime objects internally, and translates them to the end user's time zone in templates and forms.

This is handy if your users live in more than one time zone and you want to display date and time information according to each user's wall clock.

Even if your Web site is available in only one time zone, it's still good practice to store data in UTC in your database. One main reason is Daylight Saving Time (DST). Many countries have a system of DST, where clocks are moved forward in spring and backward in autumn. If you're working in local time, you're likely to encounter errors twice a year, when the transitions happen. (The pytz documentation discusses these issues in greater detail.) This probably doesn't matter for your blog, but it's a problem if you over-bill or under-bill your customers by one hour, twice a year, every year. The solution to this problem is to use UTC in the code and use local time only when interacting with end users.

Time zone support is disabled by default. To enable it, set `USE_TZ = True` in your settings file. Installing pytz is highly recommended, but not mandatory. It's as simple as:

```
$ sudo pip install pytz
```

Note: The default `settings.py` file created by `django-admin.py startproject` includes `USE_TZ = True` for convenience.

Note: There is also an independent but related `USE_L10N` setting that controls whether Django should activate format localization. See *Format localization* for more details.

If you're wrestling with a particular problem, start with the *time zone FAQ*.

Concepts

Naive and aware datetime objects

Python's `datetime.datetime` objects have a `tzinfo` attribute that can be used to store time zone information, represented as an instance of a subclass of `datetime.tzinfo`. When this attribute is set and describes an offset, a datetime object is **aware**. Otherwise, it's **naive**.

You can use `is_aware()` and `is_naive()` to determine whether datetimes are aware or naive.

When time zone support is disabled, Django uses naive datetime objects in local time. This is simple and sufficient for many use cases. In this mode, to obtain the current time, you would write:

```
import datetime

now = datetime.datetime.now()
```

When time zone support is enabled, Django uses time-zone-aware datetime objects. If your code creates datetime objects, they should be aware too. In this mode, the example above becomes:

```
import datetime
from django.utils.timezone import utc

now = datetime.datetime.utcnow().replace(tzinfo=utc)
```

Note: `django.utils.timezone` provides a `now()` function that returns a naive or aware datetime object according to the value of `USE_TZ`.

Warning: Dealing with aware datetime objects isn't always intuitive. For instance, the `tzinfo` argument of the standard datetime constructor doesn't work reliably for time zones with DST. Using UTC is generally safe; if you're using other time zones, you should review the pytz documentation carefully.

Note: Python's `datetime.time` objects also feature a `tzinfo` attribute, and PostgreSQL has a matching `time with time zone` type. However, as PostgreSQL's docs put it, this type "exhibits properties which lead to questionable usefulness".

Django only supports naive time objects and will raise an exception if you attempt to save an aware time object.

Interpretation of naive datetime objects

When `USE_TZ` is `True`, Django still accepts naive datetime objects, in order to preserve backwards-compatibility. When the database layer receives one, it attempts to make it aware by interpreting it in the *default time zone* and raises a warning.

Unfortunately, during DST transitions, some datetimes don't exist or are ambiguous. In such situations, pytz raises an exception. Other `tzinfo` implementations, such as the local time zone used as a fallback when pytz isn't installed, may raise an exception or return inaccurate results. That's why you should always create aware datetime objects when time zone support is enabled.

In practice, this is rarely an issue. Django gives you aware datetime objects in the models and forms, and most often, new datetime objects are created from existing ones through `timedelta` arithmetic. The only datetime that's often created in application code is the current time, and `timezone.now()` automatically does the right thing.

Default time zone and current time zone

The **default time zone** is the time zone defined by the `TIME_ZONE` setting.

The **current time zone** is the time zone that's used for rendering.

You should set the current time zone to the end user's actual time zone with `activate()`. Otherwise, the default time zone is used.

Note: As explained in the documentation of `TIME_ZONE`, Django sets environment variables so that its process runs in the default time zone. This happens regardless of the value of `USE_TZ` and of the current time zone.

When `USE_TZ` is `True`, this is useful to preserve backwards-compatibility with applications that still rely on local time. However, *as explained above*, this isn't entirely reliable, and you should always work with aware datetimes in UTC in your own code. For instance, use `utcfromtimestamp()` instead of `fromtimestamp()` – and don't forget to set `tzinfo` to `utc`.

Selecting the current time zone

The current time zone is the equivalent of the current *locale* for translations. However, there's no equivalent of the `Accept-Language` HTTP header that Django could use to determine the user's time zone automatically. Instead, Django provides *time zone selection functions*. Use them to build the time zone selection logic that makes sense for you.

Most Web sites that care about time zones just ask users in which time zone they live and store this information in the user's profile. For anonymous users, they use the time zone of their primary audience or UTC. pytz provides helpers, like a list of time zones per country, that you can use to pre-select the most likely choices.

Here's an example that stores the current timezone in the session. (It skips error handling entirely for the sake of simplicity.)

Add the following middleware to `MIDDLEWARE_CLASSES`:

```
from django.utils import timezone

class TimezoneMiddleware(object):
    def process_request(self, request):
        tz = request.session.get('django_timezone')
        if tz:
            timezone.activate(tz)
```

Create a view that can set the current timezone:

```
import pytz
from django.shortcuts import redirect, render

def set_timezone(request):
    if request.method == 'POST':
        request.session['django_timezone'] = pytz.timezone(request.POST['timezone'])
        return redirect('/')
    else:
        return render(request, 'template.html', {'timezones': pytz.common_timezones})
```

Include a form in `template.html` that will `POST` to this view:

```
{% load tz %}
<form action="{% url 'set_timezone' %}" method="POST">
```

```
    {% csrf_token %}
    <label for="timezone">Time zone:</label>
    <select name="timezone">
        {% for tz in timezones %}
        <option value="{{ tz }}"{% if tz == TIME_ZONE %} selected="selected"{% endif %}>{{ tz }}</opt
        {% endfor %}
    </select>
    <input type="submit" value="Set" />
</form>
```

Time zone aware input in forms

When you enable time zone support, Django interprets datetimes entered in forms in the *current time zone* and returns aware datetime objects in `cleaned_data`.

If the current time zone raises an exception for datetimes that don't exist or are ambiguous because they fall in a DST transition (the timezones provided by pytz do this), such datetimes will be reported as invalid values.

Time zone aware output in templates

When you enable time zone support, Django converts aware datetime objects to the *current time zone* when they're rendered in templates. This behaves very much like *format localization*.

> **Warning:** Django doesn't convert naive datetime objects, because they could be ambiguous, and because your code should never produce naive datetimes when time zone support is enabled. However, you can force conversion with the template filters described below.

Conversion to local time isn't always appropriate – you may be generating output for computers rather than for humans. The following filters and tags, provided by the `tz` template tag library, allow you to control the time zone conversions.

Template tags

localtime Enables or disables conversion of aware datetime objects to the current time zone in the contained block.

This tag has exactly the same effects as the `USE_TZ` setting as far as the template engine is concerned. It allows a more fine grained control of conversion.

To activate or deactivate conversion for a template block, use:

```
{% load tz %}

{% localtime on %}
    {{ value }}
{% endlocaltime %}

{% localtime off %}
    {{ value }}
{% endlocaltime %}
```

Note: The value of `USE_TZ` isn't respected inside of a `{% localtime %}` block.

---

timezone Sets or unsets the current time zone in the contained block. When the current time zone is unset, the default time zone applies.

```
{% load tz %}

{% timezone "Europe/Paris" %}
    Paris time: {{ value }}
{% endtimezone %}

{% timezone None %}
    Server time: {{ value }}
{% endtimezone %}
```

get_current_timezone When the `django.core.context_processors.tz()` context processor is enabled – by default, it is – each `RequestContext` contains a `TIME_ZONE` variable that provides the name of the current time zone.

If you don't use a `RequestContext`, you can obtain this value with the `get_current_timezone` tag:

```
{% get_current_timezone as TIME_ZONE %}
```

Template filters

These filters accept both aware and naive datetimes. For conversion purposes, they assume that naive datetimes are in the default time zone. They always return aware datetimes.

localtime Forces conversion of a single value to the current time zone.

For example:

```
{% load tz %}

{{ value|localtime }}
```

utc Forces conversion of a single value to UTC.

For example:

```
{% load tz %}

{{ value|utc }}
```

timezone Forces conversion of a single value to an arbitrary timezone.

The argument must be an instance of a `tzinfo` subclass or a time zone name. If it is a time zone name, pytz is required.

For example:

```
{% load tz %}

{{ value|timezone:"Europe/Paris" }}
```

Migration guide

Here's how to migrate a project that was started before Django supported time zones.

Database

PostgreSQL The PostgreSQL backend stores datetimes as `timestamp with time zone`. In practice, this means it converts datetimes from the connection's time zone to UTC on storage, and from UTC to the connection's time zone on retrieval.

As a consequence, if you're using PostgreSQL, you can switch between `USE_TZ = False` and `USE_TZ = True` freely. The database connection's time zone will be set to `TIME_ZONE` or `UTC` respectively, so that Django obtains correct datetimes in all cases. You don't need to perform any data conversions.

Other databases Other backends store datetimes without time zone information. If you switch from `USE_TZ = False` to `USE_TZ = True`, you must convert your data from local time to UTC – which isn't deterministic if your local time has DST.

Code

The first step is to add `USE_TZ = True` to your settings file and install pytz (if possible). At this point, things should mostly work. If you create naive datetime objects in your code, Django makes them aware when necessary.

However, these conversions may fail around DST transitions, which means you aren't getting the full benefits of time zone support yet. Also, you're likely to run into a few problems because it's impossible to compare a naive datetime with an aware datetime. Since Django now gives you aware datetimes, you'll get exceptions wherever you compare a datetime that comes from a model or a form with a naive datetime that you've created in your code.

So the second step is to refactor your code wherever you instantiate datetime objects to make them aware. This can be done incrementally. `django.utils.timezone` defines some handy helpers for compatibility code: `now()`, `is_aware()`, `is_naive()`, `make_aware()`, and `make_naive()`.

Finally, in order to help you locate code that needs upgrading, Django raises a warning when you attempt to save a naive datetime to the database:

```
RuntimeWarning: DateTimeField received a naive datetime (2012-01-01 00:00:00) while time zone support
```

During development, you can turn such warnings into exceptions and get a traceback by adding the following to your settings file:

```
import warnings
warnings.filterwarnings(
        'error', r"DateTimeField received a naive datetime",
        RuntimeWarning, r'django\.db\.models\.fields')
```

Fixtures

When serializing an aware datetime, the UTC offset is included, like this:

```
"2011-09-01T13:20:30+03:00"
```

For a naive datetime, it obviously isn't:

```
"2011-09-01T13:20:30"
```

For models with `DateTimeFields`, this difference makes it impossible to write a fixture that works both with and without time zone support.

Fixtures generated with `USE_TZ = False`, or before Django 1.4, use the "naive" format. If your project contains such fixtures, after you enable time zone support, you'll see `RuntimeWarnings` when you load them. To get rid of the warnings, you must convert your fixtures to the "aware" format.

You can regenerate fixtures with `loaddata` then `dumpdata`. Or, if they're small enough, you can simply edit them to add the UTC offset that matches your `TIME_ZONE` to each serialized datetime.

FAQ

Setup

1. **I don't need multiple time zones. Should I enable time zone support?**

   Yes. When time zone support is enabled, Django uses a more accurate model of local time. This shields you from subtle and unreproducible bugs around Daylight Saving Time (DST) transitions.

   In this regard, time zones are comparable to `unicode` in Python. At first it's hard. You get encoding and decoding errors. Then you learn the rules. And some problems disappear – you never get mangled output again when your application receives non-ASCII input.

   When you enable time zone support, you'll encounter some errors because you're using naive datetimes where Django expects aware datetimes. Such errors show up when running tests and they're easy to fix. You'll quickly learn how to avoid invalid operations.

   On the other hand, bugs caused by the lack of time zone support are much harder to prevent, diagnose and fix. Anything that involves scheduled tasks or datetime arithmetic is a candidate for subtle bugs that will bite you only once or twice a year.

   For these reasons, time zone support is enabled by default in new projects, and you should keep it unless you have a very good reason not to.

2. **I've enabled time zone support. Am I safe?**

   Maybe. You're better protected from DST-related bugs, but you can still shoot yourself in the foot by carelessly turning naive datetimes into aware datetimes, and vice-versa.

   If your application connects to other systems – for instance, if it queries a Web service – make sure datetimes are properly specified. To transmit datetimes safely, their representation should include the UTC offset, or their values should be in UTC (or both!).

   Finally, our calendar system contains interesting traps for computers:

   ```
   >>> import datetime
   >>> def one_year_before(value):        # DON'T DO THAT!
   ...     return value.replace(year=value.year - 1)
   >>> one_year_before(datetime.datetime(2012, 3, 1, 10, 0))
   datetime.datetime(2011, 3, 1, 10, 0)
   >>> one_year_before(datetime.datetime(2012, 2, 29, 10, 0))
   Traceback (most recent call last):
   ...
   ValueError: day is out of range for month
   ```

   (To implement this function, you must decide whether 2012-02-29 minus one year is 2011-02-28 or 2011-03-01, which depends on your business requirements.)

3. **Should I install pytz?**

   Yes. Django has a policy of not requiring external dependencies, and for this reason pytz is optional. However, it's much safer to install it.

   As soon as you activate time zone support, Django needs a definition of the default time zone. When pytz is available, Django loads this definition from the tz database. This is the most accurate solution. Otherwise, it relies on the difference between local time and UTC, as reported by the operating system, to compute conversions. This is less reliable, especially around DST transitions.

   Furthermore, if you want to support users in more than one time zone, pytz is the reference for time zone definitions.

Troubleshooting

1. **My application crashes with `TypeError: can't compare offset-naive and offset-aware datetimes` – what's wrong?**

   Let's reproduce this error by comparing a naive and an aware datetime:

   ```
   >>> import datetime
   >>> from django.utils import timezone
   >>> naive = datetime.datetime.utcnow()
   >>> aware = naive.replace(tzinfo=timezone.utc)
   >>> naive == aware
   Traceback (most recent call last):
   ...
   TypeError: can't compare offset-naive and offset-aware datetimes
   ```

   If you encounter this error, most likely your code is comparing these two things:

   - a datetime provided by Django – for instance, a value read from a form or a model field. Since you enabled time zone support, it's aware.
   - a datetime generated by your code, which is naive (or you wouldn't be reading this).

   Generally, the correct solution is to change your code to use an aware datetime instead.

   If you're writing a pluggable application that's expected to work independently of the value of `USE_TZ`, you may find `django.utils.timezone.now()` useful. This function returns the current date and time as a naive datetime when `USE_TZ = False` and as an aware datetime when `USE_TZ = True`. You can add or subtract `datetime.timedelta` as needed.

2. **I see lots of `RuntimeWarning: DateTimeField received a naive datetime (YYYY-MM-DD HH:MM:SS) while time zone support is active` – is that bad?**

   When time zone support is enabled, the database layer expects to receive only aware datetimes from your code. This warning occurs when it receives a naive datetime. This indicates that you haven't finished porting your code for time zone support. Please refer to the *migration guide* for tips on this process.

   In the meantime, for backwards compatibility, the datetime is considered to be in the default time zone, which is generally what you expect.

3. **`now.date()` is yesterday! (or tomorrow)**

   If you've always used naive datetimes, you probably believe that you can convert a datetime to a date by calling its `date()` method. You also consider that a `date` is a lot like a `datetime`, except that it's less accurate.

   None of this is true in a time zone aware environment:

```
>>> import datetime
>>> import pytz
>>> paris_tz = pytz.timezone("Europe/Paris")
>>> new_york_tz = pytz.timezone("America/New_York")
>>> paris = paris_tz.localize(datetime.datetime(2012, 3, 3, 1, 30))
# This is the correct way to convert between time zones with pytz.
>>> new_york = new_york_tz.normalize(paris.astimezone(new_york_tz))
>>> paris == new_york, paris.date() == new_york.date()
(True, False)
>>> paris - new_york, paris.date() - new_york.date()
(datetime.timedelta(0), datetime.timedelta(1))
>>> paris
datetime.datetime(2012, 3, 3, 1, 30, tzinfo=<DstTzInfo 'Europe/Paris' CET+1:00:00 STD>)
>>> new_york
datetime.datetime(2012, 3, 2, 19, 30, tzinfo=<DstTzInfo 'America/New_York' EST-1 day, 19:00:00 S
```

As this example shows, the same datetime has a different date, depending on the time zone in which it is represented. But the real problem is more fundamental.

A datetime represents a **point in time**. It's absolute: it doesn't depend on anything. On the contrary, a date is a **calendaring concept**. It's a period of time whose bounds depend on the time zone in which the date is considered. As you can see, these two concepts are fundamentally different, and converting a datetime to a date isn't a deterministic operation.

What does this mean in practice?

Generally, you should avoid converting a `datetime` to `date`. For instance, you can use the `date` template filter to only show the date part of a datetime. This filter will convert the datetime into the current time zone before formatting it, ensuring the results appear correctly.

If you really need to do the conversion yourself, you must ensure the datetime is converted to the appropriate time zone first. Usually, this will be the current timezone:

```
>>> from django.utils import timezone
>>> timezone.activate(pytz.timezone("Asia/Singapore"))
# For this example, we just set the time zone to Singapore, but here's how
# you would obtain the current time zone in the general case.
>>> current_tz = timezone.get_current_timezone()
# Again, this is the correct way to convert between time zones with pytz.
>>> local = current_tz.normalize(paris.astimezone(current_tz))
>>> local
datetime.datetime(2012, 3, 3, 8, 30, tzinfo=<DstTzInfo 'Asia/Singapore' SGT+8:00:00 STD>)
>>> local.date()
datetime.date(2012, 3, 3)
```

Usage

1. **I have a string `"2012-02-21 10:28:45"` and I know it's in the `"Europe/Helsinki"` time zone. How do I turn that into an aware datetime?**

   This is exactly what pytz is for.

   ```
   >>> from django.utils.dateparse import parse_datetime
   >>> naive = parse_datetime("2012-02-21 10:28:45")
   >>> import pytz
   >>> pytz.timezone("Europe/Helsinki").localize(naive, is_dst=None)
   datetime.datetime(2012, 2, 21, 10, 28, 45, tzinfo=<DstTzInfo 'Europe/Helsinki' EET+2:00:00 STD>)
   ```

Note that `localize` is a pytz extension to the `tzinfo` API. Also, you may want to catch `InvalidTimeError`. The documentation of pytz contains more examples. You should review it before attempting to manipulate aware datetimes.

2. **How can I obtain the local time in the current time zone?**

   Well, the first question is, do you really need to?

   You should only use local time when you're interacting with humans, and the template layer provides *filters and tags* to convert datetimes to the time zone of your choice.

   Furthermore, Python knows how to compare aware datetimes, taking into account UTC offsets when necessary. It's much easier (and possibly faster) to write all your model and view code in UTC. So, in most circumstances, the datetime in UTC returned by `django.utils.timezone.now()` will be sufficient.

   For the sake of completeness, though, if you really want the local time in the current time zone, here's how you can obtain it:

```
>>> from django.utils import timezone
>>> timezone.localtime(timezone.now())
datetime.datetime(2012, 3, 3, 20, 10, 53, 873365, tzinfo=<DstTzInfo 'Europe/Paris' CET+1:00:00 S
```

   In this example, pytz is installed and the current time zone is `"Europe/Paris"`.

3. **How can I see all available time zones?**

   pytz provides helpers, including a list of current time zones and a list of all available time zones – some of which are only of historical interest.

3.14.4 Overview

The goal of internationalization and localization is to allow a single Web application to offer its content in languages and formats tailored to the audience.

Django has full support for *translation of text*, *formatting of dates, times and numbers*, and *time zones*.

Essentially, Django does two things:

- It allows developers and template authors to specify which parts of their apps should be translated or formatted for local languages and cultures.
- It uses these hooks to localize Web apps for particular users according to their preferences.

Obviously, translation depends on the target language, and formatting usually depends on the target country. These informations are provided by browsers in the `Accept-Language` header. However, the time zone isn't readily available.

3.14.5 Definitions

The words "internationalization" and "localization" often cause confusion; here's a simplified definition:

internationalization Preparing the software for localization. Usually done by developers.

localization Writing the translations and local formats. Usually done by translators.

More details can be found in the W3C Web Internationalization FAQ, the Wikipedia article or the GNU gettext documentation.

Warning: Translation and formatting are controlled by `USE_I18N` and `USE_L10N` settings respectively. However, both features involve internationalization and localization. The names of the settings are an unfortunate result of Django's history.

Here are some other terms that will help us to handle a common language:

locale name A locale name, either a language specification of the form `ll` or a combined language and country specification of the form `ll_CC`. Examples: `it`, `de_AT`, `es`, `pt_BR`. The language part is always is lower case and the country part in upper case. The separator is an underscore.

language code Represents the name of a language. Browsers send the names of the languages they accept in the `Accept-Language` HTTP header using this format. Examples: `it`, `de-at`, `es`, `pt-br`. Both the language and the country parts are in lower case. The separator is a dash.

message file A message file is a plain-text file, representing a single language, that contains all available *translation strings* and how they should be represented in the given language. Message files have a `.po` file extension.

translation string A literal that can be translated.

format file A format file is a Python module that defines the data formats for a given locale.

3.15 Logging

New in version 1.3: *Please see the release notes*

3.15.1 A quick logging primer

Django uses Python's builtin `logging` module to perform system logging. The usage of this module is discussed in detail in Python's own documentation. However, if you've never used Python's logging framework (or even if you have), here's a quick primer.

The cast of players

A Python logging configuration consists of four parts:

- *Loggers*
- *Handlers*
- *Filters*
- *Formatters*

Loggers

A logger is the entry point into the logging system. Each logger is a named bucket to which messages can be written for processing.

A logger is configured to have a *log level*. This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:

- `DEBUG`: Low level system information for debugging purposes
- `INFO`: General system information
- `WARNING`: Information describing a minor problem that has occurred.

- `ERROR`: Information describing a major problem that has occurred.
- `CRITICAL`: Information describing a critical problem that has occurred.

Each message that is written to the logger is a *Log Record*. Each log record also has a *log level* indicating the severity of that specific message. A log record can also contain useful metadata that describes the event that is being logged. This can include details such as a stack trace or an error code.

When a message is given to the logger, the log level of the message is compared to the log level of the logger. If the log level of the message meets or exceeds the log level of the logger itself, the message will undergo further processing. If it doesn't, the message will be ignored.

Once a logger has determined that a message needs to be processed, it is passed to a *Handler*.

Handlers

The handler is the engine that determines what happens to each message in a logger. It describes a particular logging behavior, such as writing a message to the screen, to a file, or to a network socket.

Like loggers, handlers also have a log level. If the log level of a log record doesn't meet or exceed the level of the handler, the handler will ignore the message.

A logger can have multiple handlers, and each handler can have a different log level. In this way, it is possible to provide different forms of notification depending on the importance of a message. For example, you could install one handler that forwards `ERROR` and `CRITICAL` messages to a paging service, while a second handler logs all messages (including `ERROR` and `CRITICAL` messages) to a file for later analysis.

Filters

A filter is used to provide additional control over which log records are passed from logger to handler.

By default, any log message that meets log level requirements will be handled. However, by installing a filter, you can place additional criteria on the logging process. For example, you could install a filter that only allows `ERROR` messages from a particular source to be emitted.

Filters can also be used to modify the logging record prior to being emitted. For example, you could write a filter that downgrades `ERROR` log records to `WARNING` records if a particular set of criteria are met.

Filters can be installed on loggers or on handlers; multiple filters can be used in a chain to perform multiple filtering actions.

Formatters

Ultimately, a log record needs to be rendered as text. Formatters describe the exact format of that text. A formatter usually consists of a Python formatting string; however, you can also write custom formatters to implement specific formatting behavior.

3.15.2 Using logging

Once you have configured your loggers, handlers, filters and formatters, you need to place logging calls into your code. Using the logging framework is very simple. Here's an example:

```
# import the logging library
import logging

# Get an instance of a logger
logger = logging.getLogger(__name__)

def my_view(request, arg1, arg):
    ...
    if bad_mojo:
        # Log an error message
        logger.error('Something went wrong!')
```

And that's it! Every time the `bad_mojo` condition is activated, an error log record will be written.

Naming loggers

The call to `logging.getLogger()` obtains (creating, if necessary) an instance of a logger. The logger instance is identified by a name. This name is used to identify the logger for configuration purposes.

By convention, the logger name is usually `__name__`, the name of the python module that contains the logger. This allows you to filter and handle logging calls on a per-module basis. However, if you have some other way of organizing your logging messages, you can provide any dot-separated name to identify your logger:

```
# Get an instance of a specific named logger
logger = logging.getLogger('project.interesting.stuff')
```

The dotted paths of logger names define a hierarchy. The `project.interesting` logger is considered to be a parent of the `project.interesting.stuff` logger; the `project` logger is a parent of the `project.interesting` logger.

Why is the hierarchy important? Well, because loggers can be set to *propagate* their logging calls to their parents. In this way, you can define a single set of handlers at the root of a logger tree, and capture all logging calls in the subtree of loggers. A logging handler defined in the `project` namespace will catch all logging messages issued on the `project.interesting` and `project.interesting.stuff` loggers.

This propagation can be controlled on a per-logger basis. If you don't want a particular logger to propagate to it's parents, you can turn off this behavior.

Making logging calls

The logger instance contains an entry method for each of the default log levels:

- `logger.critical()`
- `logger.error()`
- `logger.warning()`
- `logger.info()`
- `logger.debug()`

There are two other logging calls available:

- `logger.log()`: Manually emits a logging message with a specific log level.
- `logger.exception()`: Creates an `ERROR` level logging message wrapping the current exception stack frame.

3.15.3 Configuring logging

Of course, it isn't enough to just put logging calls into your code. You also need to configure the loggers, handlers, filters and formatters to ensure that logging output is output in a useful way.

Python's logging library provides several techniques to configure logging, ranging from a programmatic interface to configuration files. By default, Django uses the dictConfig format.

Note: `logging.dictConfig` is a builtin library in Python 2.7. In order to make this library available for users of earlier Python versions, Django includes a copy as part of `django.utils.log`. If you have Python 2.7 or later, the system native library will be used; if you have Python 2.6, Django's copy will be used.

In order to configure logging, you use `LOGGING` to define a dictionary of logging settings. These settings describes the loggers, handlers, filters and formatters that you want in your logging setup, and the log levels and other properties that you want those components to have.

Logging is configured immediately after settings have been loaded. Since the loading of settings is one of the first things that Django does, you can be certain that loggers are always ready for use in your project code.

An example

The full documentation for dictConfig format is the best source of information about logging configuration dictionaries. However, to give you a taste of what is possible, here is an example of a fairly complex logging setup, configured using `logging.dictConfig()`:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level':'DEBUG',
            'class':'django.utils.log.NullHandler',
        },
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
```

```
        }
    },
    'loggers': {
        'django': {
            'handlers':['null'],
            'propagate': True,
            'level':'INFO',
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

This logging configuration does the following things:

- Identifies the configuration as being in 'dictConfig version 1' format. At present, this is the only dictConfig format version.
- Disables all existing logging configurations.
- Defines two formatters:
  - `simple`, that just outputs the log level name (e.g., `DEBUG`) and the log message.

    The *format* string is a normal Python formatting string describing the details that are to be output on each logging line. The full list of detail that can be output can be found in the formatter documentation.
  - `verbose`, that outputs the log level name, the log message, plus the time, process, thread and module that generate the log message.
- Defines one filter – `project.logging.SpecialFilter`, using the alias `special`. If this filter required additional arguments at time of construction, they can be provided as additional keys in the filter configuration dictionary. In this case, the argument `foo` will be given a value of `bar` when instantiating the `SpecialFilter`.
- Defines three handlers:
  - `null`, a NullHandler, which will pass any `DEBUG` (or higher) message to `/dev/null`.
  - `console`, a StreamHandler, which will print any `DEBUG` (or higher) message to stderr. This handler uses the *simple* output format.
  - `mail_admins`, an AdminEmailHandler, which will email any `ERROR` (or higher) message to the site admins. This handler uses the `special` filter.
- Configures three loggers:
  - `django`, which passes all messages at `INFO` or higher to the `null` handler.
  - `django.request`, which passes all `ERROR` messages to the `mail_admins` handler. In addition, this logger is marked to *not* propagate messages. This means that log messages written to `django.request` will not be handled by the `django` logger.

– `myproject.custom`, which passes all messages at `INFO` or higher that also pass the `special` filter to two handlers – the `console`, and `mail_admins`. This means that all `INFO` level messages (or higher) will be printed to the console; `ERROR` and `CRITICAL` messages will also be output via email.

Custom handlers and circular imports

If your `settings.py` specifies a custom handler class and the file defining that class also imports `settings.py` a circular import will occur.

For example, if `settings.py` contains the following config for `LOGGING`:

```
LOGGING = {
  'version': 1,
  'handlers': {
    'custom_handler': {
      'level': 'INFO',
      'class': 'myproject.logconfig.MyHandler',
    }
  }
}
```

and `myproject/logconfig.py` has the following line before the `MyHandler` definition:

```
from django.conf import settings
```

then the `dictconfig` module will raise an exception like the following:

```
ValueError: Unable to configure handler 'custom_handler':
Unable to configure handler 'custom_handler':
'module' object has no attribute 'logconfig'
```

Custom logging configuration

If you don't want to use Python's dictConfig format to configure your logger, you can specify your own configuration scheme.

The `LOGGING_CONFIG` setting defines the callable that will be used to configure Django's loggers. By default, it points at Python's `logging.dictConfig()` method. However, if you want to use a different configuration process, you can use any other callable that takes a single argument. The contents of `LOGGING` will be provided as the value of that argument when logging is configured.

Disabling logging configuration

If you don't want to configure logging at all (or you want to manually configure logging using your own approach), you can set `LOGGING_CONFIG` to `None`. This will disable the configuration process.

Note: Setting `LOGGING_CONFIG` to `None` only means that the configuration process is disabled, not logging itself. If you disable the configuration process, Django will still make logging calls, falling back to whatever default logging behavior is defined.

3.15.4 Django's logging extensions

Django provides a number of utilities to handle the unique requirements of logging in Web server environment.

Loggers

Django provides three built-in loggers.

django

`django` is the catch-all logger. No messages are posted directly to this logger.

django.request

Log messages related to the handling of requests. 5XX responses are raised as `ERROR` messages; 4XX responses are raised as `WARNING` messages.

Messages to this logger have the following extra context:

- `status_code`: The HTTP response code associated with the request.
- `request`: The request object that generated the logging message.

django.db.backends

Messages relating to the interaction of code with the database. For example, every SQL statement executed by a request is logged at the `DEBUG` level to this logger.

Messages to this logger have the following extra context:

- `duration`: The time taken to execute the SQL statement.
- `sql`: The SQL statement that was executed.
- `params`: The parameters that were used in the SQL call.

For performance reasons, SQL logging is only enabled when `settings.DEBUG` is set to `True`, regardless of the logging level or handlers that are installed.

Handlers

Django provides one log handler in addition to those provided by the Python logging module.

class **AdminEmailHandler**(\[*include_html=False*\])

This handler sends an email to the site admins for each log message it receives.

If the log record contains a `request` attribute, the full details of the request will be included in the email.

If the log record contains stack trace information, that stack trace will be included in the email.

The `include_html` argument of `AdminEmailHandler` is used to control whether the traceback email includes an HTML attachment containing the full content of the debug Web page that would have been produced if `DEBUG` were `True`. To set this value in your configuration, include it in the handler definition for `django.utils.log.AdminEmailHandler`, like this:

```
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'include_html': True,
```

```
        }
    },
```

Note that this HTML version of the email contains a full traceback, with names and values of local variables at each level of the stack, plus the values of your Django settings. This information is potentially very sensitive, and you may not want to send it over email. Consider using something such as django-sentry to get the best of both worlds – the rich information of full tracebacks plus the security of *not* sending the information over email. You may also explicitly designate certain sensitive information to be filtered out of error reports – learn more on *Filtering error reports*.

Filters

Django provides two log filters in addition to those provided by the Python logging module.

class `CallbackFilter`(*callback*)

New in version 1.4: *Please see the release notes* This filter accepts a callback function (which should accept a single argument, the record to be logged), and calls it for each record that passes through the filter. Handling of that record will not proceed if the callback returns False.

class `RequireDebugFalse`

New in version 1.4: *Please see the release notes* This filter will only pass on records when settings.DEBUG is False.

This filter is used as follows in the default `LOGGING` configuration to ensure that the `AdminEmailHandler` only sends error emails to admins when `DEBUG` is *False*:

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse',
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

3.16 Pagination

Django provides a few classes that help you manage paginated data – that is, data that's split across several pages, with "Previous/Next" links. These classes live in `django/core/paginator.py`.

3.16.1 Example

Give `Paginator` a list of objects, plus the number of items you'd like to have on each page, and it gives you methods for accessing the items for each page:

```
>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)

>>> p.count
```

```
4
>>> p.num_pages
2
>>> p.page_range
[1, 2]

>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']

>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
>>> page2.has_next()
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.next_page_number()
Traceback (most recent call last):
...
EmptyPage: That page contains no results
>>> page2.previous_page_number()
1
>>> page2.start_index() # The 1-based index of the first item on this page
3
>>> page2.end_index() # The 1-based index of the last item on this page
4

>>> p.page(0)
Traceback (most recent call last):
...
EmptyPage: That page number is less than 1
>>> p.page(3)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

Note: Note that you can give `Paginator` a list/tuple, a Django `QuerySet`, or any other object with a `count()` or `__len__()` method. When determining the number of objects contained in the passed object, `Paginator` will first try calling `count()`, then fallback to using `len()` if the passed object has no `count()` method. This allows objects such as Django's `QuerySet` to use a more efficient `count()` method when available.

3.16.2 Using `Paginator` in a view

Here's a slightly more complex example using `Paginator` in a view to paginate a queryset. We give both the view and the accompanying template to show how you can display the results. This example assumes you have a `Contacts` model that has already been imported.

The view function looks like this:

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

def listing(request):
    contact_list = Contacts.objects.all()
    paginator = Paginator(contact_list, 25) # Show 25 contacts per page

    page = request.GET.get('page')
    try:
        contacts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer, deliver first page.
        contacts = paginator.page(1)
    except EmptyPage:
        # If page is out of range (e.g. 9999), deliver last page of results.
        contacts = paginator.page(paginator.num_pages)

    return render_to_response('list.html', {"contacts": contacts})
```

In the template `list.html`, you'll want to include navigation between pages along with any interesting information from the objects themselves:

```
{% for contact in contacts %}
    {# Each "contact" is a Contact model object. #}
    {{ contact.full_name|upper }}<br />
    ...
{% endfor %}

<div class="pagination">
    <span class="step-links">
        {% if contacts.has_previous %}
            <a href="?page={{ contacts.previous_page_number }}">previous</a>
        {% endif %}

        <span class="current">
            Page {{ contacts.number }} of {{ contacts.paginator.num_pages }}.
        </span>

        {% if contacts.has_next %}
            <a href="?page={{ contacts.next_page_number }}">next</a>
        {% endif %}
    </span>
</div>
```

Changed in version 1.4: Previously, you would need to use `{% for contact in contacts.object_list %}`, since the `Page` object was not iterable.

3.16.3 `Paginator` objects

The `Paginator` class has this constructor:

class `Paginator` (*object_list*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)

Required arguments

`object_list` A list, tuple, Django `QuerySet`, or other sliceable object with a `count()` or `__len__()` method.

per_page The maximum number of items to include on a page, not including orphans (see the `orphans` optional argument below).

Optional arguments

orphans The minimum number of items allowed on the last page, defaults to zero. Use this when you don't want to have a last page with very few items. If the last page would normally have a number of items less than or equal to `orphans`, then those items will be added to the previous page (which becomes the last page) instead of leaving the items on a page by themselves. For example, with 23 items, `per_page=10`, and `orphans=3`, there will be two pages; the first page with 10 items and the second (and last) page with 13 items.

allow_empty_first_page Whether or not the first page is allowed to be empty. If `False` and `object_list` is empty, then an `EmptyPage` error will be raised.

Methods

`Paginator.`**page**(*number*)
Returns a `Page` object with the given 1-based index. Raises `InvalidPage` if the given page number doesn't exist.

Attributes

`Paginator.`**count**
The total number of objects, across all pages.

Note: When determining the number of objects contained in `object_list`, `Paginator` will first try calling `object_list.count()`. If `object_list` has no `count()` method, then `Paginator` will fallback to using `len(object_list)`. This allows objects, such as Django's `QuerySet`, to use a more efficient `count()` method when available.

`Paginator.`**num_pages**
The total number of pages.

`Paginator.`**page_range**
A 1-based range of page numbers, e.g., `[1, 2, 3, 4]`.

3.16.4 `InvalidPage` exceptions

exception InvalidPage
A base class for exceptions raised when a paginator is passed an invalid page number.

The `Paginator.page()` method raises an exception if the requested page is invalid (i.e., not an integer) or contains no objects. Generally, it's enough to trap the `InvalidPage` exception, but if you'd like more granularity, you can trap either of the following exceptions:

exception PageNotAnInteger
Raised when `page()` is given a value that isn't an integer.

exception EmptyPage
Raised when `page()` is given a valid value but no objects exist on that page.

Both of the exceptions are subclasses of `InvalidPage`, so you can handle them both with a simple `except InvalidPage`.

3.16.5 `Page` objects

You usually won't construct `Page` objects by hand – you'll get them using `Paginator.page()`.

class `Page` (*object_list*, *number*, *paginator*)

New in version 1.4: A page acts like a sequence of `Page.object_list` when using `len()` or iterating it directly.

Methods

`Page.`**`has_next`**`()`
Returns `True` if there's a next page.

`Page.`**`has_previous`**`()`
Returns `True` if there's a previous page.

`Page.`**`has_other_pages`**`()`
Returns `True` if there's a next *or* previous page.

`Page.`**`next_page_number`**`()`
Returns the next page number. Changed in version 1.5: *Please see the release notes* Raises `InvalidPage` if next page doesn't exist.

`Page.`**`previous_page_number`**`()`
Returns the previous page number. Changed in version 1.5: *Please see the release notes* Raises `InvalidPage` if previous page doesn't exist.

`Page.`**`start_index`**`()`
Returns the 1-based index of the first object on the page, relative to all of the objects in the paginator's list. For example, when paginating a list of 5 objects with 2 objects per page, the second page's `start_index()` would return `3`.

`Page.`**`end_index`**`()`
Returns the 1-based index of the last object on the page, relative to all of the objects in the paginator's list. For example, when paginating a list of 5 objects with 2 objects per page, the second page's `end_index()` would return `4`.

Attributes

`Page.`**`object_list`**
The list of objects on this page.

`Page.`**`number`**
The 1-based page number for this page.

`Page.`**`paginator`**
The associated `Paginator` object.

3.17 Python 3 compatibility

Django 1.5 is the first version of Django to support Python 3. The same code runs both on Python 2 (2.6.5) and Python 3 (3.2), thanks to the six compatibility layer and `unicode_literals`.

This document is not meant as a Python 2 to Python 3 migration guide. There are many existing resources, including Python's official porting guide. Rather, it describes guidelines that apply to Django's code and are recommended for pluggable apps that run with both Python 2 and 3.

3.17.1 Syntax requirements

Unicode

In Python 3, all strings are considered Unicode by default. The `unicode` type from Python 2 is called `str` in Python 3, and `str` becomes `bytes`.

You mustn't use the `u` prefix before a unicode string literal because it's a syntax error in Python 3.2. You must prefix byte strings with `b`.

In order to enable the same behavior in Python 2, every module must import `unicode_literals` from `__future__`:

```
from __future__ import unicode_literals

my_string = "This is an unicode literal"
my_bytestring = b"This is a bytestring"
```

Be cautious if you have to slice bytestrings.

Exceptions

When you capture exceptions, use the `as` keyword:

```
try:
    ...
except MyException as exc:
    ...
```

This older syntax was removed in Python 3:

```
try:
    ...
except MyException, exc:
    ...
```

The syntax to reraise an exception with a different traceback also changed. Use `six.reraise()`.

3.17.2 Writing compatible code with six

six is the canonical compatibility library for supporting Python 2 and 3 in a single codebase. Read its documentation!

`six` is bundled with Django: you can import it as `django.utils.six`.

Here are the most common changes required to write compatible code.

String types

The `basestring` and `unicode` types were removed in Python 3, and the meaning of `str` changed. To test these types, use the following idioms:

```
isinstance(myvalue, six.string_types)       # replacement for basestring
isinstance(myvalue, six.text_type)          # replacement for unicode
isinstance(myvalue, bytes)                  # replacement for str
```

Python 2.6 provides `bytes` as an alias for `str`, so you don't need `six.binary_type`.

long

The `long` type no longer exists in Python 3. `1L` is a syntax error. Use `six.integer_types` check if a value is an integer or a long:

```
isinstance(myvalue, six.integer_types)      # replacement for (int, long)
```

xrange

Import `six.moves.xrange()` wherever you use `xrange`.

Moved modules

Some modules were renamed in Python 3. The `django.utils.six.moves` module provides a compatible location to import them.

In addition to six' defaults, Django's version provides `dummy_thread` as `_dummy_thread`.

PY3

If you need different code in Python 2 and Python 3, check `six.PY3`:

```
if six.PY3:
    # do stuff Python 3-wise
else:
    # do stuff Python 2-wise
```

This is a last resort solution when `six` doesn't provide an appropriate function.

3.17.3 Customizations of six

The version of six bundled with Django includes a few additional tools:

iterlists(*MultiValueDict*)
Returns an iterator over the lists of values of a `MultiValueDict`. This replaces `iterlists()` on Python 2 and `lists()` on Python 3.

3.18 Security in Django

This document is an overview of Django's security features. It includes advice on securing a Django-powered site.

3.18.1 Cross site scripting (XSS) protection

XSS attacks allow a user to inject client side scripts into the browsers of other users. This is usually achieved by storing the malicious scripts in the database where it will be retrieved and displayed to other users, or by getting users to click a link which will cause the attacker's JavaScript to be executed by the user's browser. However, XSS attacks can originate from any untrusted source of data, such as cookies or Web services, whenever the data is not sufficiently sanitized before including in a page.

Using Django templates protects you against the majority of XSS attacks. However, it is important to understand what protections it provides and its limitations.

Django templates *escape specific characters* which are particularly dangerous to HTML. While this protects users from most malicious input, it is not entirely foolproof. For example, it will not protect the following:

```
<style class={{ var }}>...</style>
```

If `var` is set to `'class1 onmouseover=javascript:func()'`, this can result in unauthorized JavaScript execution, depending on how the browser renders imperfect HTML.

It is also important to be particularly careful when using `is_safe` with custom template tags, the `safe` template tag, `mark_safe`, and when autoescape is turned off.

In addition, if you are using the template system to output something other than HTML, there may be entirely separate characters and words which require escaping.

You should also be very careful when storing HTML in the database, especially when that HTML is retrieved and displayed.

Markup library

If you use `django.contrib.markup`, you need to ensure that the filters are only used on trusted input, or that you have correctly configured them to ensure they do not allow raw HTML output. See the documentation of that module for more information.

3.18.2 Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent.

Django has built-in protection against most types of CSRF attacks, providing you have *enabled and used it* where appropriate. However, as with any mitigation technique, there are limitations. For example, it is possible to disable the CSRF module globally or for particular views. You should only do this if you know what you are doing. There are other *limitations* if your site has subdomains that are outside of your control.

CSRF protection works by checking for a nonce in each POST request. This ensures that a malicious user cannot simply "replay" a form POST to your Web site and have another logged in user unwittingly submit that form. The malicious user would have to know the nonce, which is user specific (using a cookie).

Be very careful with marking views with the `csrf_exempt` decorator unless it is absolutely necessary.

3.18.3 SQL injection protection

SQL injection is a type of attack where a malicious user is able to execute arbitrary SQL code on a database. This can result in records being deleted or data leakage.

By using Django's querysets, the resulting SQL will be properly escaped by the underlying database driver. However, Django also gives developers power to write *raw queries* or execute *custom sql*. These capabilities should be used sparingly and you should always be careful to properly escape any parameters that the user can control. In addition, you should exercise caution when using `extra()`.

3.18.4 Clickjacking protection

Clickjacking is a type of attack where a malicious site wraps another site in a frame. This attack can result in an unsuspecting user being tricked into performing unintended actions on the target site.

Django contains *clickjacking protection* in the form of the `X-Frame-Options middleware` which in a supporting browser can prevent a site from being rendered inside a frame. It is possible to disable the protection on a per view basis or to configure the exact header value sent.

The middleware is strongly recommended for any site that does not need to have its pages wrapped in a frame by third party sites, or only needs to allow that for a small section of the site.

3.18.5 SSL/HTTPS

It is always better for security, though not always practical in all cases, to deploy your site behind HTTPS. Without this, it is possible for malicious network users to sniff authentication credentials or any other information transferred between client and server, and in some cases – **active** network attackers – to alter data that is sent in either direction.

If you want the protection that HTTPS provides, and have enabled it on your server, there are some additional steps you may need:

- If necessary, set `SECURE_PROXY_SSL_HEADER`, ensuring that you have understood the warnings there thoroughly. Failure to do this can result in CSRF vulnerabilities, and failure to do it correctly can also be dangerous!
- Set up redirection so that requests over HTTP are redirected to HTTPS.

  This could be done using a custom middleware. Please note the caveats under `SECURE_PROXY_SSL_HEADER`. For the case of a reverse proxy, it may be easier or more secure to configure the main Web server to do the redirect to HTTPS.
- Use 'secure' cookies.

  If a browser connects initially via HTTP, which is the default for most browsers, it is possible for existing cookies to be leaked. For this reason, you should set your `SESSION_COOKIE_SECURE` and `CSRF_COOKIE_SECURE` settings to `True`. This instructs the browser to only send these cookies over HTTPS connections. Note that this will mean that sessions will not work over HTTP, and the CSRF protection will prevent any POST data being accepted over HTTP (which will be fine if you are redirecting all HTTP traffic to HTTPS).

3.18.6 Host headers and virtual hosting

Django uses the `Host` header provided by the client to construct URLs in certain cases. While these values are sanitized to prevent Cross Site Scripting attacks, they can be used for Cross-Site Request Forgery and cache poisoning attacks in some circumstances. We recommend you ensure your Web server is configured such that:

- It always validates incoming HTTP `Host` headers against the expected host name.
- Disallows requests with no `Host` header.
- Is *not* configured with a catch-all virtual host that forwards requests to a Django application.

Additionally, as of 1.3.1, Django requires you to explicitly enable support for the `X-Forwarded-Host` header if your configuration requires it.

3.18.7 Additional security topics

While Django provides good security protection out of the box, it is still important to properly deploy your application and take advantage of the security protection of the Web server, operating system and other components.

- Make sure that your Python code is outside of the Web server's root. This will ensure that your Python code is not accidentally served as plain text (or accidentally executed).
- Take care with any *user uploaded files*.

- Django does not throttle requests to authenticate users. To protect against brute-force attacks against the authentication system, you may consider deploying a Django plugin or Web server module to throttle these requests.
- If your site accepts file uploads, it is strongly advised that you limit these uploads in your Web server configuration to a reasonable size in order to prevent denial of service (DOS) attacks. In Apache, this can be easily set using the LimitRequestBody directive.
- Keep your `SECRET_KEY` a secret.
- It is a good idea to limit the accessibility of your caching system and database using a firewall.

3.19 Serializing Django objects

Django's serialization framework provides a mechanism for "translating" Django models into other formats. Usually these other formats will be text-based and used for sending Django data over a wire, but it's possible for a serializer to handle any format (text-based or not).

See Also:

If you just want to get some data from your tables into a serialized form, you could use the `dumpdata` management command.

3.19.1 Serializing data

At the highest level, serializing data is a very simple operation:

```
from django.core import serializers
data = serializers.serialize("xml", SomeModel.objects.all())
```

The arguments to the `serialize` function are the format to serialize the data to (see Serialization formats) and a `QuerySet` to serialize. (Actually, the second argument can be any iterator that yields Django model instances, but it'll almost always be a QuerySet).

You can also use a serializer object directly:

```
XMLSerializer = serializers.get_serializer("xml")
xml_serializer = XMLSerializer()
xml_serializer.serialize(queryset)
data = xml_serializer.getvalue()
```

This is useful if you want to serialize data directly to a file-like object (which includes an `HttpResponse`):

```
with open("file.xml", "w") as out:
    xml_serializer.serialize(SomeModel.objects.all(), stream=out)
```

Note: Calling `get_serializer()` with an unknown *format* will raise a `SerializerDoesNotExist` exception.

Subset of fields

If you only want a subset of fields to be serialized, you can specify a `fields` argument to the serializer:

```
from django.core import serializers
data = serializers.serialize('xml', SomeModel.objects.all(), fields=('name','size'))
```

In this example, only the `name` and `size` attributes of each model will be serialized.

Note: Depending on your model, you may find that it is not possible to deserialize a model that only serializes a subset of its fields. If a serialized object doesn't specify all the fields that are required by a model, the deserializer will not be able to save deserialized instances.

Inherited Models

If you have a model that is defined using an *abstract base class*, you don't have to do anything special to serialize that model. Just call the serializer on the object (or objects) that you want to serialize, and the output will be a complete representation of the serialized object.

However, if you have a model that uses *multi-table inheritance*, you also need to serialize all of the base classes for the model. This is because only the fields that are locally defined on the model will be serialized. For example, consider the following models:

```
class Place(models.Model):
    name = models.CharField(max_length=50)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField()
```

If you only serialize the Restaurant model:

```
data = serializers.serialize('xml', Restaurant.objects.all())
```

the fields on the serialized output will only contain the *serves_hot_dogs* attribute. The *name* attribute of the base class will be ignored.

In order to fully serialize your Restaurant instances, you will need to serialize the Place models as well:

```
all_objects = list(Restaurant.objects.all()) + list(Place.objects.all())
data = serializers.serialize('xml', all_objects)
```

3.19.2 Deserializing data

Deserializing data is also a fairly simple operation:

```
for obj in serializers.deserialize("xml", data):
    do_something_with(obj)
```

As you can see, the `deserialize` function takes the same format argument as `serialize`, a string or stream of data, and returns an iterator.

However, here it gets slightly complicated. The objects returned by the `deserialize` iterator *aren't* simple Django objects. Instead, they are special `DeserializedObject` instances that wrap a created – but unsaved – object and any associated relationship data.

Calling `DeserializedObject.save()` saves the object to the database.

This ensures that deserializing is a non-destructive operation even if the data in your serialized representation doesn't match what's currently in the database. Usually, working with these `DeserializedObject` instances looks something like:

```
for deserialized_object in serializers.deserialize("xml", data):
    if object_should_be_saved(deserialized_object):
        deserialized_object.save()
```

In other words, the usual use is to examine the deserialized objects to make sure that they are "appropriate" for saving before doing so. Of course, if you trust your data source you could just save the object and move on.

The Django object itself can be inspected as `deserialized_object.object`.

3.19.3 Serialization formats

Django supports a number of serialization formats, some of which require you to install third-party Python modules:

Identifier	Information
`xml`	Serializes to and from a simple XML dialect.
`json`	Serializes to and from JSON.
`yaml`	Serializes to YAML (YAML Ain't a Markup Language). This serializer is only available if PyYAML is installed.

Notes for specific serialization formats

json

If you're using UTF-8 (or any other non-ASCII encoding) data with the JSON serializer, you must pass `ensure_ascii=False` as a parameter to the `serialize()` call. Otherwise, the output won't be encoded correctly.

For example:

```
json_serializer = serializers.get_serializer("json")()
json_serializer.serialize(queryset, ensure_ascii=False, stream=response)
```

Be aware that not all Django output can be passed unmodified to `json`. In particular, *lazy translation objects* need a special encoder written for them. Something like this will work:

```
import json
from django.utils.functional import Promise
from django.utils.encoding import force_unicode

class LazyEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Promise):
            return force_unicode(obj)
        return super(LazyEncoder, self).default(obj)
```

3.19.4 Natural keys

The default serialization strategy for foreign keys and many-to-many relations is to serialize the value of the primary key(s) of the objects in the relation. This strategy works well for most objects, but it can cause difficulty in some circumstances.

Consider the case of a list of objects that have a foreign key referencing `ContentType`. If you're going to serialize an object that refers to a content type, then you need to have a way to refer to that content type to begin with. Since `ContentType` objects are automatically created by Django during the database synchronization process, the primary key of a given content type isn't easy to predict; it will depend on how and when `syncdb` was executed. This is true for all models which automatically generate objects, notably including `Permission`, `Group`, and `User`.

Warning: You should never include automatically generated objects in a fixture or other serialized data. By chance, the primary keys in the fixture may match those in the database and loading the fixture will have no effect. In the more likely case that they don't match, the fixture loading will fail with an `IntegrityError`.

There is also the matter of convenience. An integer id isn't always the most convenient way to refer to an object; sometimes, a more natural reference would be helpful.

It is for these reasons that Django provides *natural keys*. A natural key is a tuple of values that can be used to uniquely identify an object instance without using the primary key value.

Deserialization of natural keys

Consider the following two models:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    class Meta:
        unique_together = (('first_name', 'last_name'),)

class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person)
```

Ordinarily, serialized data for `Book` would use an integer to refer to the author. For example, in JSON, a Book might be serialized as:

```
...
{
    "pk": 1,
    "model": "store.book",
    "fields": {
        "name": "Mostly Harmless",
        "author": 42
    }
}
...
```

This isn't a particularly natural way to refer to an author. It requires that you know the primary key value for the author; it also requires that this primary key value is stable and predictable.

However, if we add natural key handling to Person, the fixture becomes much more humane. To add natural key handling, you define a default Manager for Person with a `get_by_natural_key()` method. In the case of a Person, a good natural key might be the pair of first and last name:

```
from django.db import models

class PersonManager(models.Manager):
    def get_by_natural_key(self, first_name, last_name):
        return self.get(first_name=first_name, last_name=last_name)

class Person(models.Model):
    objects = PersonManager()
```

```
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    class Meta:
        unique_together = (('first_name', 'last_name'),)
```

Now books can use that natural key to refer to `Person` objects:

```
...
{
    "pk": 1,
    "model": "store.book",
    "fields": {
        "name": "Mostly Harmless",
        "author": ["Douglas", "Adams"]
    }
}
...
```

When you try to load this serialized data, Django will use the `get_by_natural_key()` method to resolve `["Douglas", "Adams"]` into the primary key of an actual `Person` object.

Note: Whatever fields you use for a natural key must be able to uniquely identify an object. This will usually mean that your model will have a uniqueness clause (either unique=True on a single field, or `unique_together` over multiple fields) for the field or fields in your natural key. However, uniqueness doesn't need to be enforced at the database level. If you are certain that a set of fields will be effectively unique, you can still use those fields as a natural key.

Serialization of natural keys

So how do you get Django to emit a natural key when serializing an object? Firstly, you need to add another method – this time to the model itself:

```
class Person(models.Model):
    objects = PersonManager()

    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)

    birthdate = models.DateField()

    def natural_key(self):
        return (self.first_name, self.last_name)

    class Meta:
        unique_together = (('first_name', 'last_name'),)
```

That method should always return a natural key tuple – in this example, `(first name, last name)`. Then, when you call `serializers.serialize()`, you provide a `use_natural_keys=True` argument:

```
>>> serializers.serialize('json', [book1, book2], indent=2, use_natural_keys=True)
```

When `use_natural_keys=True` is specified, Django will use the `natural_key()` method to serialize any reference to objects of the type that defines the method.

If you are using `dumpdata` to generate serialized data, you use the *–natural* command line flag to generate natural keys.

Note: You don't need to define both `natural_key()` and `get_by_natural_key()`. If you don't want Django to output natural keys during serialization, but you want to retain the ability to load natural keys, then you can opt to not implement the `natural_key()` method.

Conversely, if (for some strange reason) you want Django to output natural keys during serialization, but *not* be able to load those key values, just don't define the `get_by_natural_key()` method.

Dependencies during serialization

Since natural keys rely on database lookups to resolve references, it is important that the data exists before it is referenced. You can't make a *forward reference* with natural keys – the data you're referencing must exist before you include a natural key reference to that data.

To accommodate this limitation, calls to `dumpdata` that use the `--natural` option will serialize any model with a `natural_key()` method before serializing standard primary key objects.

However, this may not always be enough. If your natural key refers to another object (by using a foreign key or natural key to another object as part of a natural key), then you need to be able to ensure that the objects on which a natural key depends occur in the serialized data before the natural key requires them.

To control this ordering, you can define dependencies on your `natural_key()` methods. You do this by setting a `dependencies` attribute on the `natural_key()` method itself.

For example, let's add a natural key to the `Book` model from the example above:

```
class Book(models.Model):
    name = models.CharField(max_length=100)
    author = models.ForeignKey(Person)

    def natural_key(self):
        return (self.name,) + self.author.natural_key()
```

The natural key for a `Book` is a combination of its name and its author. This means that `Person` must be serialized before `Book`. To define this dependency, we add one extra line:

```
def natural_key(self):
    return (self.name,) + self.author.natural_key()
natural_key.dependencies = ['example_app.person']
```

This definition ensures that all `Person` objects are serialized before any `Book` objects. In turn, any object referencing `Book` will be serialized after both `Person` and `Book` have been serialized.

3.20 Django settings

A Django settings file contains all the configuration of your Django installation. This document explains how settings work and which settings are available.

3.20.1 The basics

A settings file is just a Python module with module-level variables.

Here are a couple of example settings:

```
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
TEMPLATE_DIRS = ('/home/templates/mike', '/home/templates/john')
```

Because a settings file is a Python module, the following apply:

- It doesn't allow for Python syntax errors.
- It can assign settings dynamically using normal Python syntax. For example:

  ```
  MY_SETTING = [str(i) for i in range(30)]
  ```

- It can import values from other settings files.

3.20.2 Designating the settings

When you use Django, you have to tell it which settings you're using. Do this by using an environment variable, `DJANGO_SETTINGS_MODULE`.

The value of `DJANGO_SETTINGS_MODULE` should be in Python path syntax, e.g. `mysite.settings`. Note that the settings module should be on the Python import search path.

The django-admin.py utility

When using *django-admin.py*, you can either set the environment variable once, or explicitly pass in the settings module each time you run the utility.

Example (Unix Bash shell):

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin.py runserver
```

Example (Windows shell):

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin.py runserver
```

Use the `--settings` command-line argument to specify the settings manually:

```
django-admin.py runserver --settings=mysite.settings
```

On the server (mod_wsgi)

In your live server environment, you'll need to tell your WSGI application what settings file to use. Do that with `os.environ`:

```
import os

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

Read the *Django mod_wsgi documentation* for more information and other common elements to a Django WSGI application.

3.20.3 Default settings

A Django settings file doesn't have to define any settings if it doesn't need to. Each setting has a sensible default value. These defaults live in the module `django/conf/global_settings.py`.

Here's the algorithm Django uses in compiling settings:

- Load settings from `global_settings.py`.
- Load settings from the specified settings file, overriding the global settings as necessary.

Note that a settings file should *not* import from `global_settings`, because that's redundant.

Seeing which settings you've changed

There's an easy way to view which of your settings deviate from the default settings. The command `python manage.py diffsettings` displays differences between the current settings file and Django's default settings.

For more, see the `diffsettings` documentation.

3.20.4 Using settings in Python code

In your Django apps, use settings by importing the object `django.conf.settings`. Example:

```
from django.conf import settings

if settings.DEBUG:
    # Do something
```

Note that `django.conf.settings` isn't a module – it's an object. So importing individual settings is not possible:

```
from django.conf.settings import DEBUG  # This won't work.
```

Also note that your code should *not* import from either `global_settings` or your own settings file. `django.conf.settings` abstracts the concepts of default settings and site-specific settings; it presents a single interface. It also decouples the code that uses settings from the location of your settings.

3.20.5 Altering settings at runtime

You shouldn't alter settings in your applications at runtime. For example, don't do this in a view:

```
from django.conf import settings

settings.DEBUG = True   # Don't do this!
```

The only place you should assign to settings is in a settings file.

3.20.6 Security

Because a settings file contains sensitive information, such as the database password, you should make every attempt to limit access to it. For example, change its file permissions so that only you and your Web server's user can read it. This is especially important in a shared-hosting environment.

3.20.7 Available settings

For a full list of available settings, see the *settings reference*.

3.20.8 Creating your own settings

There's nothing stopping you from creating your own settings, for your own Django apps. Just follow these conventions:

- Setting names are in all uppercase.
- Don't reinvent an already-existing setting.

For settings that are sequences, Django itself uses tuples, rather than lists, but this is only a convention.

3.20.9 Using settings without setting DJANGO_SETTINGS_MODULE

In some cases, you might want to bypass the `DJANGO_SETTINGS_MODULE` environment variable. For example, if you're using the template system by itself, you likely don't want to have to set up an environment variable pointing to a settings module.

In these cases, you can configure Django's settings manually. Do this by calling:

`django.conf.settings.`**`configure`**`(`*default_settings*, ***settings*`)`

Example:

```
from django.conf import settings

settings.configure(DEBUG=True, TEMPLATE_DEBUG=True,
    TEMPLATE_DIRS=('/home/web-apps/myapp', '/home/web-apps/base'))
```

Pass `configure()` as many keyword arguments as you'd like, with each keyword argument representing a setting and its value. Each argument name should be all uppercase, with the same name as the settings described above. If a particular setting is not passed to `configure()` and is needed at some later point, Django will use the default setting value.

Configuring Django in this fashion is mostly necessary – and, indeed, recommended – when you're using a piece of the framework inside a larger application.

Consequently, when configured via `settings.configure()`, Django will not make any modifications to the process environment variables (see the documentation of `TIME_ZONE` for why this would normally occur). It's assumed that you're already in full control of your environment in these cases.

Custom default settings

If you'd like default values to come from somewhere other than `django.conf.global_settings`, you can pass in a module or class that provides the default settings as the `default_settings` argument (or as the first positional argument) in the call to `configure()`.

In this example, default settings are taken from `myapp_defaults`, and the `DEBUG` setting is set to `True`, regardless of its value in `myapp_defaults`:

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

The following example, which uses `myapp_defaults` as a positional argument, is equivalent:

```
settings.configure(myapp_defaults, DEBUG=True)
```

Normally, you will not need to override the defaults in this fashion. The Django defaults are sufficiently tame that you can safely use them. Be aware that if you do pass in a new default module, it entirely *replaces* the Django defaults, so you must specify a value for every possible setting that might be used in that code you are importing. Check in `django.conf.settings.global_settings` for the full list.

Either configure() or DJANGO_SETTINGS_MODULE is required

If you're not setting the `DJANGO_SETTINGS_MODULE` environment variable, you *must* call `configure()` at some point before using any code that reads settings.

If you don't set `DJANGO_SETTINGS_MODULE` and don't call `configure()`, Django will raise an `ImportError` exception the first time a setting is accessed.

If you set `DJANGO_SETTINGS_MODULE`, access settings values somehow, *then* call `configure()`, Django will raise a `RuntimeError` indicating that settings have already been configured. There is a property just for this purpose:

For example:

```
from django.conf import settings
if not settings.configured:
    settings.configure(myapp_defaults, DEBUG=True)
```

Also, it's an error to call `configure()` more than once, or to call `configure()` after any setting has been accessed.

It boils down to this: Use exactly one of either `configure()` or `DJANGO_SETTINGS_MODULE`. Not both, and not neither.

3.21 Signals

Django includes a "signal dispatcher" which helps allow decoupled applications get notified when actions occur elsewhere in the framework. In a nutshell, signals allow certain *senders* to notify a set of *receivers* that some action has taken place. They're especially useful when many pieces of code may be interested in the same events.

Django provides a *set of built-in signals* that let user code get notified by Django itself of certain actions. These include some useful notifications:

- `django.db.models.signals.pre_save` & `django.db.models.signals.post_save`

  Sent before or after a model's `save()` method is called.

- `django.db.models.signals.pre_delete` & `django.db.models.signals.post_delete`

  Sent before or after a model's `delete()` method or queryset's `delete()` method is called.

- `django.db.models.signals.m2m_changed`

  Sent when a `ManyToManyField` on a model is changed.

- `django.core.signals.request_started` & `django.core.signals.request_finished`

  Sent when Django starts or finishes an HTTP request.

See the *built-in signal documentation* for a complete list, and a complete explanation of each signal.

You can also define and send your own custom signals; see below.

3.21.1 Listening to signals

To receive a signal, you need to register a *receiver* function that gets called when the signal is sent by using the `Signal.connect()` method:

`Signal.`**`connect`**`(`*receiver*[, *sender=None*, *weak=True*, *dispatch_uid=None*]`)`

Parameters

- **receiver** – The callback function which will be connected to this signal. See *Receiver functions* for more information.
- **sender** – Specifies a particular sender to receive signals from. See *Connecting to signals sent by specific senders* for more information.
- **weak** – Django stores signal handlers as weak references by default. Thus, if your receiver is a local function, it may be garbage collected. To prevent this, pass `weak=False` when you call the signal's `connect()` method.
- **dispatch_uid** – A unique identifier for a signal receiver in cases where duplicate signals may be sent. See *Preventing duplicate signals* for more information.

Let's see how this works by registering a signal that gets called after each HTTP request is finished. We'll be connecting to the `request_finished` signal.

Receiver functions

First, we need to define a receiver function. A receiver can be any Python function or method:

```
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Notice that the function takes a `sender` argument, along with wildcard keyword arguments (`**kwargs`); all signal handlers must take these arguments.

We'll look at senders a bit later, but right now look at the `**kwargs` argument. All signals send keyword arguments, and may change those keyword arguments at any time. In the case of `request_finished`, it's documented as sending no arguments, which means we might be tempted to write our signal handling as `my_callback(sender)`. This would be wrong – in fact, Django will throw an error if you do so. That's because at any point arguments could get added to the signal and your receiver must be able to handle those new arguments.

Connecting receiver functions

There are two ways you can connect a receiver to a signal. You can take the manual connect route:

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

Alternatively, you can use a `receiver` decorator when you define your receiver:

```
from django.core.signals import request_finished
from django.dispatch import receiver

@receiver(request_finished)
def my_callback(sender, **kwargs):
    print("Request finished!")
```

Now, our `my_callback` function will be called each time a request finishes.

Note that `receiver` can also take a list of signals to connect a function to. New in version 1.3: *Please see the release notes* The `receiver` decorator was added in Django 1.3. Changed in version 1.5: *Please see the release notes* The ability to pass a list of signals was added.

Where should this code live?

You can put signal handling and registration code anywhere you like. However, you'll need to make sure that the module it's in gets imported early on so that the signal handling gets registered before any signals need to be sent. This makes your app's `models.py` a good place to put registration of signal handlers.

Connecting to signals sent by specific senders

Some signals get sent many times, but you'll only be interested in receiving a certain subset of those signals. For example, consider the `django.db.models.signals.pre_save` signal sent before a model gets saved. Most of the time, you don't need to know when *any* model gets saved – just when one *specific* model is saved.

In these cases, you can register to receive signals sent only by particular senders. In the case of `django.db.models.signals.pre_save`, the sender will be the model class being saved, so you can indicate that you only want signals sent by some model:

```
from django.db.models.signals import pre_save
from django.dispatch import receiver
from myapp.models import MyModel

@receiver(pre_save, sender=MyModel)
def my_handler(sender, **kwargs):
    ...
```

The `my_handler` function will only be called when an instance of `MyModel` is saved.

Different signals use different objects as their senders; you'll need to consult the *built-in signal documentation* for details of each particular signal.

Preventing duplicate signals

In some circumstances, the module in which you are connecting signals may be imported multiple times. This can cause your receiver function to be registered more than once, and thus called multiples times for a single signal event.

If this behavior is problematic (such as when using signals to send an email whenever a model is saved), pass a unique identifier as the `dispatch_uid` argument to identify your receiver function. This identifier will usually be a string, although any hashable object will suffice. The end result is that your receiver function will only be bound to the signal once for each unique `dispatch_uid` value.

```
from django.core.signals import request_finished

request_finished.connect(my_callback, dispatch_uid="my_unique_identifier")
```

3.21.2 Defining and sending signals

Your applications can take advantage of the signal infrastructure and provide its own signals.

Defining signals

class `Signal` ([*providing_args=list*])

All signals are `django.dispatch.Signal` instances. The `providing_args` is a list of the names of arguments the signal will provide to listeners.

For example:

```
import django.dispatch

pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])
```

This declares a `pizza_done` signal that will provide receivers with `toppings` and `size` arguments.

Remember that you're allowed to change this list of arguments at any time, so getting the API right on the first try isn't necessary.

Sending signals

There are two ways to send signals in Django.

`Signal.`**`send`**(*sender*, ***kwargs*)

`Signal.`**`send_robust`**(*sender*, ***kwargs*)

To send a signal, call either `Signal.send()` or `Signal.send_robust()`. You must provide the `sender` argument, and may provide as many other keyword arguments as you like.

For example, here's how sending our `pizza_done` signal might look:

```
class PizzaStore(object):
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self, toppings=toppings, size=size)
        ...
```

Both `send()` and `send_robust()` return a list of tuple pairs `[(receiver, response), ... ]`, representing the list of called receiver functions and their response values.

`send()` differs from `send_robust()` in how exceptions raised by receiver functions are handled. `send()` does *not* catch any exceptions raised by receivers; it simply allows errors to propagate. Thus not all receivers may be notified of a signal in the face of an error.

`send_robust()` catches all errors derived from Python's `Exception` class, and ensures all receivers are notified of the signal. If an error occurs, the error instance is returned in the tuple pair for the receiver that raised the error.

3.21.3 Disconnecting signals

`Signal.`**`disconnect`**([*receiver=None*, *sender=None*, *weak=True*, *dispatch_uid=None*])

To disconnect a receiver from a signal, call `Signal.disconnect()`. The arguments are as described in `Signal.connect()`.

The *receiver* argument indicates the registered receiver to disconnect. It may be `None` if `dispatch_uid` is used to identify the receiver.

CHAPTER

FOUR

"HOW-TO" GUIDES

Here you'll find short answers to "How do I....?" types of questions. These how-to guides don't cover topics in depth – you'll find that material in the *Using Django* and the *API Reference*. However, these guides will help you quickly accomplish common tasks.

4.1 Authenticating against Django's user database from Apache

Since keeping multiple authentication databases in sync is a common problem when dealing with Apache, you can configuring Apache to authenticate against Django's *authentication system* directly. This requires Apache version >= 2.2 and mod_wsgi >= 2.0. For example, you could:

- Serve static/media files directly from Apache only to authenticated users.
- Authenticate access to a Subversion repository against Django users with a certain permission.
- Allow certain users to connect to a WebDAV share created with mod_dav.

4.1.1 Configuring Apache

To check against Django's authorization database from a Apache configuration file, you'll need to set 'wsgi' as the value of `AuthBasicProvider` or `AuthDigestProvider` directive and then use the `WSGIAuthUserScript` directive to set the path to your authentification script:

```
<Location /example/>
    AuthType Basic
    AuthName "example.com"
    AuthBasicProvider wsgi
    WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
    Require valid-user
</Location>
```

Your auth.wsgi script will have to implement either a `check_password(environ, user, password)` function (for `AuthBasicProvider`) or a `get_realm_hash(environ, user, realm)` function (for `AuthDigestProvider`).

See the mod_wsgi documentation for more details about the implementation of such a solution.

4.2 Authentication using REMOTE_USER

This document describes how to make use of external authentication sources (where the Web server sets the REMOTE_USER environment variable) in your Django applications. This type of authentication solution is typically seen on intranet sites, with single sign-on solutions such as IIS and Integrated Windows Authentication or Apache and mod_authnz_ldap, CAS, Cosign, WebAuth, mod_auth_sspi, etc.

When the Web server takes care of authentication it typically sets the REMOTE_USER environment variable for use in the underlying application. In Django, REMOTE_USER is made available in the request.META attribute. Django can be configured to make use of the REMOTE_USER value using the RemoteUserMiddleware and RemoteUserBackend classes found in django.contrib.auth.

4.2.1 Configuration

First, you must add the django.contrib.auth.middleware.RemoteUserMiddleware to the MIDDLEWARE_CLASSES setting **after** the django.contrib.auth.middleware.AuthenticationMiddleware:

```
MIDDLEWARE_CLASSES = (
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
    ...
    )
```

Next, you must replace the ModelBackend with RemoteUserBackend in the AUTHENTICATION_BACKENDS setting:

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.RemoteUserBackend',
)
```

With this setup, RemoteUserMiddleware will detect the username in request.META['REMOTE_USER'] and will authenticate and auto-login that user using the RemoteUserBackend.

Note: Since the RemoteUserBackend inherits from ModelBackend, you will still have all of the same permissions checking that is implemented in ModelBackend.

If your authentication mechanism uses a custom HTTP header and not REMOTE_USER, you can subclass RemoteUserMiddleware and set the header attribute to the desired request.META key. For example:

```
from django.contrib.auth.middleware import RemoteUserMiddleware

class CustomHeaderMiddleware(RemoteUserMiddleware):
    header = 'HTTP_AUTHUSER'
```

4.2.2 RemoteUserBackend

class django.contrib.auth.backends.**RemoteUserBackend**

If you need more control, you can create your own authentication backend that inherits from RemoteUserBackend and overrides certain parts:

Attributes

`RemoteUserBackend.`**`create_unknown_user`**
: `True` or `False`. Determines whether or not a `User` object is created if not already in the database. Defaults to `True`.

Methods

`RemoteUserBackend.`**`clean_username`**(*username*)
: Performs any cleaning on the `username` (e.g. stripping LDAP DN information) prior to using it to get or create a `User` object. Returns the cleaned username.

`RemoteUserBackend.`**`configure_user`**(*user*)
: Configures a newly created user. This method is called immediately after a new user is created, and can be used to perform custom setup actions, such as setting the user's groups based on attributes in an LDAP directory. Returns the user object.

4.3 Writing custom django-admin commands

Applications can register their own actions with `manage.py`. For example, you might want to add a `manage.py` action for a Django app that you're distributing. In this document, we will be building a custom `closepoll` command for the `polls` application from the *tutorial*.

To do this, just add a `management/commands` directory to the application. Django will register a `manage.py` command for each Python module in that directory whose name doesn't begin with an underscore. For example:

```
polls/
    __init__.py
    models.py
    management/
        __init__.py
        commands/
            __init__.py
            _private.py
            closepoll.py
    tests.py
    views.py
```

In this example, the `closepoll` command will be made available to any project that includes the `polls` application in `INSTALLED_APPS`.

The `_private.py` module will not be available as a management command.

The `closepoll.py` module has only one requirement – it must define a class `Command` that extends `BaseCommand` or one of its *subclasses*.

Standalone scripts

Custom management commands are especially useful for running standalone scripts or for scripts that are periodically executed from the UNIX crontab or from Windows scheduled tasks control panel.

To implement the command, edit `polls/management/commands/closepoll.py` to look like this:

```
from django.core.management.base import BaseCommand, CommandError
from polls.models import Poll

class Command(BaseCommand):
    args = '<poll_id poll_id ...>'
    help = 'Closes the specified poll for voting'

    def handle(self, *args, **options):
        for poll_id in args:
            try:
                poll = Poll.objects.get(pk=int(poll_id))
            except Poll.DoesNotExist:
                raise CommandError('Poll "%s" does not exist' % poll_id)

            poll.opened = False
            poll.save()

            self.stdout.write('Successfully closed poll "%s"' % poll_id)
```

Note: When you are using management commands and wish to provide console output, you should write to `self.stdout` and `self.stderr`, instead of printing to `stdout` and `stderr` directly. By using these proxies, it becomes much easier to test your custom command.

The new custom command can be called using `python manage.py closepoll <poll_id>`.

The `handle()` method takes zero or more `poll_ids` and sets `poll.opened` to `False` for each one. If the user referenced any nonexistent polls, a `CommandError` is raised. The `poll.opened` attribute does not exist in the *tutorial* and was added to `polls.models.Poll` for this example.

The same `closepoll` could be easily modified to delete a given poll instead of closing it by accepting additional command line options. These custom options must be added to `option_list` like this:

```
from optparse import make_option

class Command(BaseCommand):
    option_list = BaseCommand.option_list + (
        make_option('--delete',
            action='store_true',
            dest='delete',
            default=False,
            help='Delete poll instead of closing it'),
        )

    def handle(self, *args, **options):
        # ...
        if options['delete']:
            poll.delete()
        # ...
```

The option (`delete` in our example) is available in the options dict parameter of the handle method. See the `optparse` Python documentation for more about `make_option` usage.

In addition to being able to add custom command line options, all *management commands* can accept some default options such as `--verbosity` and `--traceback`.

Management commands and locales

The `BaseCommand.execute()` method sets the hardcoded `en-us` locale because the commands shipped with

Django perform several tasks (for example, user-facing content rendering and database population) that require a system-neutral string language (for which we use `en-us`).

If your custom management command uses another locale, you should manually activate and deactivate it in your `handle()` or `handle_noargs()` method using the functions provided by the I18N support code:

```
from django.core.management.base import BaseCommand, CommandError
from django.utils import translation

class Command(BaseCommand):
    ...
    can_import_settings = True

    def handle(self, *args, **options):

        # Activate a fixed locale, e.g. Russian
        translation.activate('ru')

        # Or you can activate the LANGUAGE_CODE
        # chosen in the settings:
        #
        #from django.conf import settings
        #translation.activate(settings.LANGUAGE_CODE)

        # Your command logic here
        # ...

        translation.deactivate()
```

Take into account though, that system management commands typically have to be very careful about running in non-uniform locales, so:

- Make sure the `USE_I18N` setting is always `True` when running the command (this is one good example of the potential problems stemming from a dynamic runtime environment that Django commands avoid offhand by always using a fixed locale).
- Review the code of your command and the code it calls for behavioral differences when locales are changed and evaluate its impact on predictable behavior of your command.

4.3.1 Command objects

class `BaseCommand`

The base class from which all management commands ultimately derive.

Use this class if you want access to all of the mechanisms which parse the command-line arguments and work out what code to call in response; if you don't need to change any of that behavior, consider using one of its *subclasses*.

Subclassing the `BaseCommand` class requires that you implement the `handle()` method.

Attributes

All attributes can be set in your derived class and can be used in `BaseCommand`'s *subclasses*.

`BaseCommand.`**`args`**
A string listing the arguments accepted by the command, suitable for use in help messages; e.g., a command which takes a list of application names might set this to '<appname appname ...>'.

BaseCommand.**can_import_settings**
: A boolean indicating whether the command needs to be able to import Django settings; if `True`, `execute()` will verify that this is possible before proceeding. Default value is `True`.

BaseCommand.**help**
: A short description of the command, which will be printed in the help message when the user runs the command `python manage.py help <command>`.

BaseCommand.**option_list**
: This is the list of `optparse` options which will be fed into the command's `OptionParser` for parsing arguments.

BaseCommand.**output_transaction**
: A boolean indicating whether the command outputs SQL statements; if `True`, the output will automatically be wrapped with `BEGIN;` and `COMMIT;`. Default value is `False`.

BaseCommand.**requires_model_validation**
: A boolean; if `True`, validation of installed models will be performed prior to executing the command. Default value is `True`. To validate an individual application's models rather than all applications' models, call `validate()` from `handle()`.

Methods

`BaseCommand` has a few methods that can be overridden but only the `handle()` method must be implemented.

Implementing a constructor in a subclass

If you implement `__init__` in your subclass of `BaseCommand`, you must call `BaseCommand`'s `__init__`.

```
class Command(BaseCommand):
    def __init__(self, *args, **kwargs):
        super(Command, self).__init__(*args, **kwargs)
        # ...
```

BaseCommand.**get_version**()
: Return the Django version, which should be correct for all built-in Django commands. User-supplied commands can override this method to return their own version.

BaseCommand.**execute**(**args*, ***options*)
: Try to execute this command, performing model validation if needed (as controlled by the attribute `requires_model_validation`). If the command raises a `CommandError`, intercept it and print it sensibly to stderr.

Calling a management command in your code

`execute()` should not be called directly from your code to execute a command. Use *call_command* instead.

BaseCommand.**handle**(**args*, ***options*)
: The actual logic of the command. Subclasses must implement this method.

BaseCommand subclasses

class AppCommand

A management command which takes one or more installed application names as arguments, and does something with each of them.

Rather than implementing `handle()`, subclasses must implement `handle_app()`, which will be called once for each application.

`AppCommand.`**`handle_app`**(*app*, ***options*)
Perform the command's actions for `app`, which will be the Python module corresponding to an application name given on the command line.

class `LabelCommand`

A management command which takes one or more arbitrary arguments (labels) on the command line, and does something with each of them.

Rather than implementing `handle()`, subclasses must implement `handle_label()`, which will be called once for each label.

`LabelCommand.`**`handle_label`**(*label*, ***options*)
Perform the command's actions for `label`, which will be the string as given on the command line.

class `NoArgsCommand`

A command which takes no arguments on the command line.

Rather than implementing `handle()`, subclasses must implement `handle_noargs()`; `handle()` itself is overridden to ensure no arguments are passed to the command.

`NoArgsCommand.`**`handle_noargs`**(***options*)
Perform this command's actions

Command exceptions

class `CommandError`

Exception class indicating a problem while executing a management command.

If this exception is raised during the execution of a management command from a command line console, it will be caught and turned into a nicely-printed error message to the appropriate output stream (i.e., stderr); as a result, raising this exception (with a sensible description of the error) is the preferred way to indicate that something has gone wrong in the execution of a command.

If a management command is called from code through *call_command*, it's up to you to catch the exception when needed.

4.4 Writing custom model fields

4.4.1 Introduction

The *model reference* documentation explains how to use Django's standard field classes – `CharField`, `DateField`, etc. For many purposes, those classes are all you'll need. Sometimes, though, the Django version won't meet your precise requirements, or you'll want to use a field that is entirely different from those shipped with Django.

Django's built-in field types don't cover every possible database column type – only the common types, such as `VARCHAR` and `INTEGER`. For more obscure column types, such as geographic polygons or even user-created types such as PostgreSQL custom types, you can define your own Django `Field` subclasses.

Alternatively, you may have a complex Python object that can somehow be serialized to fit into a standard database column type. This is another case where a `Field` subclass will help you use your object with your models.

Our example object

Creating custom fields requires a bit of attention to detail. To make things easier to follow, we'll use a consistent example throughout this document: wrapping a Python object representing the deal of cards in a hand of Bridge. Don't worry, you don't have know how to play Bridge to follow this example. You only need to know that 52 cards are dealt out equally to four players, who are traditionally called *north*, *east*, *south* and *west*. Our class looks something like this:

```
class Hand(object):
    """A hand of cards (bridge style)"""

    def __init__(self, north, east, south, west):
        # Input parameters are lists of cards ('Ah', '9s', etc)
        self.north = north
        self.east = east
        self.south = south
        self.west = west

    # ... (other possibly useful methods omitted) ...
```

This is just an ordinary Python class, with nothing Django-specific about it. We'd like to be able to do things like this in our models (we assume the `hand` attribute on the model is an instance of `Hand`):

```
example = MyModel.objects.get(pk=1)
print(example.hand.north)

new_hand = Hand(north, east, south, west)
example.hand = new_hand
example.save()
```

We assign to and retrieve from the `hand` attribute in our model just like any other Python class. The trick is to tell Django how to handle saving and loading such an object.

In order to use the `Hand` class in our models, we **do not** have to change this class at all. This is ideal, because it means you can easily write model support for existing classes where you cannot change the source code.

Note: You might only be wanting to take advantage of custom database column types and deal with the data as standard Python types in your models; strings, or floats, for example. This case is similar to our `Hand` example and we'll note any differences as we go along.

4.4.2 Background theory

Database storage

The simplest way to think of a model field is that it provides a way to take a normal Python object – string, boolean, `datetime`, or something more complex like `Hand` – and convert it to and from a format that is useful when dealing with the database (and serialization, but, as we'll see later, that falls out fairly naturally once you have the database side under control).

Fields in a model must somehow be converted to fit into an existing database column type. Different databases provide different sets of valid column types, but the rule is still the same: those are the only types you have to work with. Anything you want to store in the database must fit into one of those types.

Normally, you're either writing a Django field to match a particular database column type, or there's a fairly straightforward way to convert your data to, say, a string.

For our `Hand` example, we could convert the card data to a string of 104 characters by concatenating all the cards together in a pre-determined order – say, all the *north* cards first, then the *east*, *south* and *west* cards. So `Hand` objects can be saved to text or character columns in the database.

What does a field class do?

class `Field`

All of Django's fields (and when we say *fields* in this document, we always mean model fields and not *form fields*) are subclasses of `django.db.models.Field`. Most of the information that Django records about a field is common to all fields – name, help text, uniqueness and so forth. Storing all that information is handled by `Field`. We'll get into the precise details of what `Field` can do later on; for now, suffice it to say that everything descends from `Field` and then customizes key pieces of the class behavior.

It's important to realize that a Django field class is not what is stored in your model attributes. The model attributes contain normal Python objects. The field classes you define in a model are actually stored in the `Meta` class when the model class is created (the precise details of how this is done are unimportant here). This is because the field classes aren't necessary when you're just creating and modifying attributes. Instead, they provide the machinery for converting between the attribute value and what is stored in the database or sent to the *serializer*.

Keep this in mind when creating your own custom fields. The Django `Field` subclass you write provides the machinery for converting between your Python instances and the database/serializer values in various ways (there are differences between storing a value and using a value for lookups, for example). If this sounds a bit tricky, don't worry – it will become clearer in the examples below. Just remember that you will often end up creating two classes when you want a custom field:

- The first class is the Python object that your users will manipulate. They will assign it to the model attribute, they will read from it for displaying purposes, things like that. This is the `Hand` class in our example.
- The second class is the `Field` subclass. This is the class that knows how to convert your first class back and forth between its permanent storage form and the Python form.

4.4.3 Writing a field subclass

When planning your `Field` subclass, first give some thought to which existing `Field` class your new field is most similar to. Can you subclass an existing Django field and save yourself some work? If not, you should subclass the `Field` class, from which everything is descended.

Initializing your new field is a matter of separating out any arguments that are specific to your case from the common arguments and passing the latter to the `__init__()` method of `Field` (or your parent class).

In our example, we'll call our field `HandField`. (It's a good idea to call your `Field` subclass `<Something>Field`, so it's easily identifiable as a `Field` subclass.) It doesn't behave like any existing field, so we'll subclass directly from `Field`:

```
from django.db import models

class HandField(models.Field):

    description = "A hand of cards (bridge style)"

    def __init__(self, *args, **kwargs):
        kwargs['max_length'] = 104
        super(HandField, self).__init__(*args, **kwargs)
```

Our `HandField` accepts most of the standard field options (see the list below), but we ensure it has a fixed length, since it only needs to hold 52 card values plus their suits; 104 characters in total.

Note: Many of Django's model fields accept options that they don't do anything with. For example, you can pass both `editable` and `auto_now` to a `django.db.models.DateField` and it will simply ignore the `editable` parameter (`auto_now` being set implies `editable=False`). No error is raised in this case.

This behavior simplifies the field classes, because they don't need to check for options that aren't necessary. They just pass all the options to the parent class and then don't use them later on. It's up to you whether you want your fields to be more strict about the options they select, or to use the simpler, more permissive behavior of the current fields.

`Field.__init__()`

The `__init__()` method takes the following parameters:

- `verbose_name`
- `name`
- `primary_key`
- `max_length`
- `unique`
- `blank`
- `null`
- `db_index`
- `rel`: Used for related fields (like `ForeignKey`). For advanced use only.
- `default`
- `editable`
- `serialize`: If `False`, the field will not be serialized when the model is passed to Django's *serializers*. Defaults to `True`.
- `unique_for_date`
- `unique_for_month`
- `unique_for_year`
- `choices`
- `help_text`
- `db_column`
- `db_tablespace`: Only for index creation, if the backend supports *tablespaces*. You can usually ignore this option.
- `auto_created`: True if the field was automatically created, as for the *OneToOneField* used by model inheritance. For advanced use only.

All of the options without an explanation in the above list have the same meaning they do for normal Django fields. See the *field documentation* for examples and details.

The `SubfieldBase` metaclass

class `django.db.models.`**`SubfieldBase`**

As we indicated in the introduction, field subclasses are often needed for two reasons: either to take advantage of a custom database column type, or to handle complex Python types. Obviously, a combination of the two is also possible. If you're only working with custom database column types and your model fields appear in Python as standard Python types direct from the database backend, you don't need to worry about this section.

If you're handling custom Python types, such as our `Hand` class, we need to make sure that when Django initializes an instance of our model and assigns a database value to our custom field attribute, we convert that value into the appropriate Python object. The details of how this happens internally are a little complex, but the code you need to write in your `Field` class is simple: make sure your field subclass uses a special metaclass:

For example:

```
class HandField(models.Field):

    description = "A hand of cards (bridge style)"

    __metaclass__ = models.SubfieldBase

    def __init__(self, *args, **kwargs):
        # ...
```

This ensures that the `to_python()` method, documented below, will always be called when the attribute is initialized.

ModelForms and custom fields

If you use `SubfieldBase`, `to_python()` will be called every time an instance of the field is assigned a value. This means that whenever a value may be assigned to the field, you need to ensure that it will be of the correct datatype, or that you handle any exceptions.

This is especially important if you use *ModelForms*. When saving a ModelForm, Django will use form values to instantiate model instances. However, if the cleaned form data can't be used as valid input to the field, the normal form validation process will break.

Therefore, you must ensure that the form field used to represent your custom field performs whatever input validation and data cleaning is necessary to convert user-provided form input into a *to_python()*-compatible model field value. This may require writing a custom form field, and/or implementing the `formfield()` method on your field to return a form field class whose *to_python()* returns the correct datatype.

Documenting your custom field

`Field.`**`description`**

As always, you should document your field type, so users will know what it is. In addition to providing a docstring for it, which is useful for developers, you can also allow users of the admin app to see a short description of the field type via the *django.contrib.admindocs* application. To do this simply provide descriptive text in a `description` class attribute of your custom field. In the above example, the description displayed by the `admindocs` application for a `HandField` will be 'A hand of cards (bridge style)'.

Useful methods

Once you've created your `Field` subclass and set up the `__metaclass__`, you might consider overriding a few standard methods, depending on your field's behavior. The list of methods below is in approximately decreasing order of importance, so start from the top.

Custom database types

`Field.db_type(self, connection)`

Returns the database column data type for the `Field`, taking into account the connection object, and the settings associated with it.

Say you've created a PostgreSQL custom type called `mytype`. You can use this field with Django by subclassing `Field` and implementing the `db_type()` method, like so:

```
from django.db import models

class MytypeField(models.Field):
    def db_type(self, connection):
        return 'mytype'
```

Once you have `MytypeField`, you can use it in any model, just like any other `Field` type:

```
class Person(models.Model):
    name = models.CharField(max_length=80)
    something_else = MytypeField()
```

If you aim to build a database-agnostic application, you should account for differences in database column types. For example, the date/time column type in PostgreSQL is called `timestamp`, while the same column in MySQL is called `datetime`. The simplest way to handle this in a `db_type()` method is to check the `connection.settings_dict['ENGINE']` attribute.

For example:

```
class MyDateField(models.Field):
    def db_type(self, connection):
        if connection.settings_dict['ENGINE'] == 'django.db.backends.mysql':
            return 'datetime'
        else:
            return 'timestamp'
```

The `db_type()` method is only called by Django when the framework constructs the `CREATE TABLE` statements for your application – that is, when you first create your tables. It's not called at any other time, so it can afford to execute slightly complex code, such as the `connection.settings_dict` check in the above example.

Some database column types accept parameters, such as `CHAR(25)`, where the parameter `25` represents the maximum column length. In cases like these, it's more flexible if the parameter is specified in the model rather than being hard-coded in the `db_type()` method. For example, it wouldn't make much sense to have a `CharMaxlength25Field`, shown here:

```
# This is a silly example of hard-coded parameters.
class CharMaxlength25Field(models.Field):
    def db_type(self, connection):
        return 'char(25)'

# In the model:
class MyModel(models.Model):
    # ...
    my_field = CharMaxlength25Field()
```

The better way of doing this would be to make the parameter specifiable at run time – i.e., when the class is instantiated. To do that, just implement `django.db.models.Field.__init__()`, like so:

```
# This is a much more flexible example.
class BetterCharField(models.Field):
```

```
    def __init__(self, max_length, *args, **kwargs):
        self.max_length = max_length
        super(BetterCharField, self).__init__(*args, **kwargs)

    def db_type(self, connection):
        return 'char(%s)' % self.max_length

# In the model:
class MyModel(models.Model):
    # ...
    my_field = BetterCharField(25)
```

Finally, if your column requires truly complex SQL setup, return `None` from `db_type()`. This will cause Django's SQL creation code to skip over this field. You are then responsible for creating the column in the right table in some other way, of course, but this gives you a way to tell Django to get out of the way.

Converting database values to Python objects

`Field.to_python(self, value)`

Converts a value as returned by your database (or a serializer) to a Python object.

The default implementation simply returns `value`, for the common case in which the database backend already returns data in the correct format (as a Python string, for example).

If your custom `Field` class deals with data structures that are more complex than strings, dates, integers or floats, then you'll need to override this method. As a general rule, the method should deal gracefully with any of the following arguments:

- An instance of the correct type (e.g., `Hand` in our ongoing example).
- A string (e.g., from a deserializer).
- Whatever the database returns for the column type you're using.

In our `HandField` class, we're storing the data as a VARCHAR field in the database, so we need to be able to process strings and `Hand` instances in `to_python()`:

```
import re

class HandField(models.Field):
    # ...

    def to_python(self, value):
        if isinstance(value, Hand):
            return value

        # The string case.
        p1 = re.compile('.{26}')
        p2 = re.compile('..')
        args = [p2.findall(x) for x in p1.findall(value)]
        if len(args) != 4:
            raise ValidationError("Invalid input for a Hand instance")
        return Hand(*args)
```

Notice that we always return a `Hand` instance from this method. That's the Python object type we want to store in the model's attribute. If anything is going wrong during value conversion, you should raise a `ValidationError` exception.

Remember: If your custom field needs the `to_python()` method to be called when it is created, you should be using The SubfieldBase metaclass mentioned earlier. Otherwise `to_python()` won't be called automatically.

Converting Python objects to query values

`Field.`**`get_prep_value`**(*self*, *value*)

This is the reverse of `to_python()` when working with the database backends (as opposed to serialization). The `value` parameter is the current value of the model's attribute (a field has no reference to its containing model, so it cannot retrieve the value itself), and the method should return data in a format that has been prepared for use as a parameter in a query.

This conversion should *not* include any database-specific conversions. If database-specific conversions are required, they should be made in the call to `get_db_prep_value()`.

For example:

```
class HandField(models.Field):
    # ...

    def get_prep_value(self, value):
        return ''.join([''.join(l) for l in (value.north,
                value.east, value.south, value.west)])
```

Converting query values to database values

`Field.`**`get_db_prep_value`**(*self*, *value*, *connection*, *prepared=False*)

Some data types (for example, dates) need to be in a specific format before they can be used by a database backend. `get_db_prep_value()` is the method where those conversions should be made. The specific connection that will be used for the query is passed as the `connection` parameter. This allows you to use backend-specific conversion logic if it is required.

The `prepared` argument describes whether or not the value has already been passed through `get_prep_value()` conversions. When `prepared` is False, the default implementation of `get_db_prep_value()` will call `get_prep_value()` to do initial data conversions before performing any database-specific processing.

`Field.`**`get_db_prep_save`**(*self*, *value*, *connection*)

Same as the above, but called when the Field value must be *saved* to the database. As the default implementation just calls `get_db_prep_value()`, you shouldn't need to implement this method unless your custom field needs a special conversion when being saved that is not the same as the conversion used for normal query parameters (which is implemented by `get_db_prep_value()`).

Preprocessing values before saving

`Field.`**`pre_save`**(*self*, *model_instance*, *add*)

This method is called just prior to `get_db_prep_save()` and should return the value of the appropriate attribute from `model_instance` for this field. The attribute name is in `self.attname` (this is set up by `Field`). If the model is being saved to the database for the first time, the `add` parameter will be `True`, otherwise it will be `False`.

You only need to override this method if you want to preprocess the value somehow, just before saving. For example, Django's `DateTimeField` uses this method to set the attribute correctly in the case of `auto_now` or `auto_now_add`.

If you do override this method, you must return the value of the attribute at the end. You should also update the model's attribute if you make any changes to the value so that code holding references to the model will always see the correct value.

Preparing values for use in database lookups

As with value conversions, preparing a value for database lookups is a two phase process.

`Field.`**`get_prep_lookup`**(*self*, *lookup_type*, *value*)

`get_prep_lookup()` performs the first phase of lookup preparation, performing generic data validity checks

Prepares the `value` for passing to the database when used in a lookup (a `WHERE` constraint in SQL). The `lookup_type` will be one of the valid Django filter lookups: `exact`, `iexact`, `contains`, `icontains`, `gt`, `gte`, `lt`, `lte`, `in`, `startswith`, `istartswith`, `endswith`, `iendswith`, `range`, `year`, `month`, `day`, `isnull`, `search`, `regex`, and `iregex`.

Your method must be prepared to handle all of these `lookup_type` values and should raise either a `ValueError` if the `value` is of the wrong sort (a list when you were expecting an object, for example) or a `TypeError` if your field does not support that type of lookup. For many fields, you can get by with handling the lookup types that need special handling for your field and pass the rest to the `get_db_prep_lookup()` method of the parent class.

If you needed to implement `get_db_prep_save()`, you will usually need to implement `get_prep_lookup()`. If you don't, `get_prep_value` will be called by the default implementation, to manage `exact`, `gt`, `gte`, `lt`, `lte`, `in` and `range` lookups.

You may also want to implement this method to limit the lookup types that could be used with your custom field type.

Note that, for `range` and `in` lookups, `get_prep_lookup` will receive a list of objects (presumably of the right type) and will need to convert them to a list of things of the right type for passing to the database. Most of the time, you can reuse `get_prep_value()`, or at least factor out some common pieces.

For example, the following code implements `get_prep_lookup` to limit the accepted lookup types to `exact` and `in`:

```
class HandField(models.Field):
    # ...

    def get_prep_lookup(self, lookup_type, value):
        # We only handle 'exact' and 'in'. All others are errors.
        if lookup_type == 'exact':
            return self.get_prep_value(value)
        elif lookup_type == 'in':
            return [self.get_prep_value(v) for v in value]
        else:
            raise TypeError('Lookup type %r not supported.' % lookup_type)
```

`Field.`**`get_db_prep_lookup`**(*self*, *lookup_type*, *value*, *connection*, *prepared=False*)

Performs any database-specific data conversions required by a lookup. As with `get_db_prep_value()`, the specific connection that will be used for the query is passed as the `connection` parameter. The `prepared` argument describes whether the value has already been prepared with `get_prep_lookup()`.

Specifying the form field for a model field

`Field.`**`formfield`**(*self*, *form_class=forms.CharField*, ***kwargs*)

Returns the default form field to use when this field is displayed in a model. This method is called by the `ModelForm` helper.

All of the `kwargs` dictionary is passed directly to the form field's `Field__init__()` method. Normally, all you need to do is set up a good default for the `form_class` argument and then delegate further handling to the parent class. This might require you to write a custom form field (and even a form widget). See the *forms documentation* for information about this, and take a look at the code in `django.contrib.localflavor` for some examples of custom widgets.

Continuing our ongoing example, we can write the `formfield()` method as:

```
class HandField(models.Field):
    # ...

    def formfield(self, **kwargs):
        # This is a fairly standard way to set up some defaults
        # while letting the caller override them.
        defaults = {'form_class': MyFormField}
        defaults.update(kwargs)
        return super(HandField, self).formfield(**defaults)
```

This assumes we've imported a `MyFormField` field class (which has its own default widget). This document doesn't cover the details of writing custom form fields.

Emulating built-in field types

`Field.`**`get_internal_type`**`(`*`self`*`)`

Returns a string giving the name of the `Field` subclass we are emulating at the database level. This is used to determine the type of database column for simple cases.

If you have created a `db_type()` method, you don't need to worry about `get_internal_type()` – it won't be used much. Sometimes, though, your database storage is similar in type to some other field, so you can use that other field's logic to create the right column.

For example:

```
class HandField(models.Field):
    # ...

    def get_internal_type(self):
        return 'CharField'
```

No matter which database backend we are using, this will mean that `syncdb` and other SQL commands create the right column type for storing a string.

If `get_internal_type()` returns a string that is not known to Django for the database backend you are using – that is, it doesn't appear in `django.db.backends.<db_name>.creation.DATA_TYPES` – the string will still be used by the serializer, but the default `db_type()` method will return `None`. See the documentation of `db_type()` for reasons why this might be useful. Putting a descriptive string in as the type of the field for the serializer is a useful idea if you're ever going to be using the serializer output in some other place, outside of Django.

Converting field data for serialization

`Field.`**`value_to_string`**`(`*`self`*`,` *`obj`*`)`

This method is used by the serializers to convert the field into a string for output. Calling `Field._get_val_from_obj(obj)()` is the best way to get the value to serialize. For example, since our `HandField` uses strings for its data storage anyway, we can reuse some existing conversion code:

```
class HandField(models.Field):
    # ...

    def value_to_string(self, obj):
        value = self._get_val_from_obj(obj)
        return self.get_prep_value(value)
```

Some general advice

Writing a custom field can be a tricky process, particularly if you're doing complex conversions between your Python types and your database and serialization formats. Here are a couple of tips to make things go more smoothly:

1. Look at the existing Django fields (in `django/db/models/fields/__init__.py`) for inspiration. Try to find a field that's similar to what you want and extend it a little bit, instead of creating an entirely new field from scratch.
2. Put a `__str__()` or `__unicode__()` method on the class you're wrapping up as a field. There are a lot of places where the default behavior of the field code is to call `force_unicode()` on the value. (In our examples in this document, `value` would be a `Hand` instance, not a `HandField`). So if your `__unicode__()` method automatically converts to the string form of your Python object, you can save yourself a lot of work.

4.4.4 Writing a `FileField` subclass

In addition to the above methods, fields that deal with files have a few other special requirements which must be taken into account. The majority of the mechanics provided by `FileField`, such as controlling database storage and retrieval, can remain unchanged, leaving subclasses to deal with the challenge of supporting a particular type of file.

Django provides a `File` class, which is used as a proxy to the file's contents and operations. This can be subclassed to customize how the file is accessed, and what methods are available. It lives at `django.db.models.fields.files`, and its default behavior is explained in the *file documentation*.

Once a subclass of `File` is created, the new `FileField` subclass must be told to use it. To do so, simply assign the new `File` subclass to the special `attr_class` attribute of the `FileField` subclass.

A few suggestions

In addition to the above details, there are a few guidelines which can greatly improve the efficiency and readability of the field's code.

1. The source for Django's own `ImageField` (in `django/db/models/fields/files.py`) is a great example of how to subclass `FileField` to support a particular type of file, as it incorporates all of the techniques described above.
2. Cache file attributes wherever possible. Since files may be stored in remote storage systems, retrieving them may cost extra time, or even money, that isn't always necessary. Once a file is retrieved to obtain some data about its content, cache as much of that data as possible to reduce the number of times the file must be retrieved on subsequent calls for that information.

4.5 Custom template tags and filters

Django's template system comes with a wide variety of *built-in tags and filters* designed to address the presentation logic needs of your application. Nevertheless, you may find yourself needing functionality that is not covered by the core set of template primitives. You can extend the template engine by defining custom tags and filters using Python, and then make them available to your templates using the `{% load %}` tag.

4.5.1 Code layout

Custom template tags and filters must live inside a Django app. If they relate to an existing app it makes sense to bundle them there; otherwise, you should create a new app to hold them.

The app should contain a `templatetags` directory, at the same level as `models.py`, `views.py`, etc. If this doesn't already exist, create it - don't forget the `__init__.py` file to ensure the directory is treated as a Python package.

Your custom tags and filters will live in a module inside the `templatetags` directory. The name of the module file is the name you'll use to load the tags later, so be careful to pick a name that won't clash with custom tags and filters in another app.

For example, if your custom tags/filters are in a file called `poll_extras.py`, your app layout might look like this:

```
polls/
    models.py
    templatetags/
        __init__.py
        poll_extras.py
    views.py
```

And in your template you would use the following:

```
{% load poll_extras %}
```

The app that contains the custom tags must be in `INSTALLED_APPS` in order for the `{% load %}` tag to work. This is a security feature: It allows you to host Python code for many template libraries on a single host machine without enabling access to all of them for every Django installation.

There's no limit on how many modules you put in the `templatetags` package. Just keep in mind that a `{% load %}` statement will load tags/filters for the given Python module name, not the name of the app.

To be a valid tag library, the module must contain a module-level variable named `register` that is a `template.Library` instance, in which all the tags and filters are registered. So, near the top of your module, put the following:

```
from django import template

register = template.Library()
```

Behind the scenes

For a ton of examples, read the source code for Django's default filters and tags. They're in `django/template/defaultfilters.py` and `django/template/defaulttags.py`, respectively.

For more information on the `load` tag, read its documentation.

4.5.2 Writing custom template filters

Custom filters are just Python functions that take one or two arguments:

- The value of the variable (input) – not necessarily a string.
- The value of the argument – this can have a default value, or be left out altogether.

For example, in the filter `{{ var|foo:"bar" }}`, the filter `foo` would be passed the variable `var` and the argument `"bar"`.

Filter functions should always return something. They shouldn't raise exceptions. They should fail silently. In case of error, they should return either the original input or an empty string – whichever makes more sense.

Here's an example filter definition:

```
def cut(value, arg):
    """Removes all values of arg from the given string"""
    return value.replace(arg, '')
```

And here's an example of how that filter would be used:

```
{{ somevariable|cut:"0" }}
```

Most filters don't take arguments. In this case, just leave the argument out of your function. Example:

```
def lower(value): # Only one argument.
    """Converts a string into all lowercase"""
    return value.lower()
```

Registering custom filters

Once you've written your filter definition, you need to register it with your `Library` instance, to make it available to Django's template language:

```
register.filter('cut', cut)
register.filter('lower', lower)
```

The `Library.filter()` method takes two arguments:

1. The name of the filter – a string.
2. The compilation function – a Python function (not the name of the function as a string).

You can use `register.filter()` as a decorator instead:

```
@register.filter(name='cut')
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
def lower(value):
    return value.lower()
```

If you leave off the `name` argument, as in the second example above, Django will use the function's name as the filter name.

Finally, `register.filter()` also accepts three keyword arguments, `is_safe`, `needs_autoescape`, and `expects_localtime`. These arguments are described in *filters and auto-escaping* and *filters and time zones* below.

Template filters that expect strings

If you're writing a template filter that only expects a string as the first argument, you should use the decorator `stringfilter`. This will convert an object to its string value before being passed to your function:

```
from django import template
from django.template.defaultfilters import stringfilter

register = template.Library()

@register.filter
@stringfilter
def lower(value):
    return value.lower()
```

This way, you'll be able to pass, say, an integer to this filter, and it won't cause an `AttributeError` (because integers don't have `lower()` methods).

Filters and auto-escaping

When writing a custom filter, give some thought to how the filter will interact with Django's auto-escaping behavior. Note that three types of strings can be passed around inside the template code:

- **Raw strings** are the native Python `str` or `unicode` types. On output, they're escaped if auto-escaping is in effect and presented unchanged, otherwise.
- **Safe strings** are strings that have been marked safe from further escaping at output time. Any necessary escaping has already been done. They're commonly used for output that contains raw HTML that is intended to be interpreted as-is on the client side.

  Internally, these strings are of type `SafeString` or `SafeUnicode`. They share a common base class of `SafeData`, so you can test for them using code like:

  ```
  if isinstance(value, SafeData):
      # Do something with the "safe" string.
      ...
  ```

- **Strings marked as "needing escaping"** are *always* escaped on output, regardless of whether they are in an `autoescape` block or not. These strings are only escaped once, however, even if auto-escaping applies.

  Internally, these strings are of type `EscapeString` or `EscapeUnicode`. Generally you don't have to worry about these; they exist for the implementation of the `escape` filter.

Template filter code falls into one of two situations:

1. Your filter does not introduce any HTML-unsafe characters (`<`, `>`, `'`, `"` or `&`) into the result that were not already present. In this case, you can let Django take care of all the auto-escaping handling for you. All you need to do is set the `is_safe` flag to `True` when you register your filter function, like so:

   ```
   @register.filter(is_safe=True)
   def myfilter(value):
       return value
   ```

   This flag tells Django that if a "safe" string is passed into your filter, the result will still be "safe" and if a non-safe string is passed in, Django will automatically escape it, if necessary.

   You can think of this as meaning "this filter is safe – it doesn't introduce any possibility of unsafe HTML."

The reason `is_safe` is necessary is because there are plenty of normal string operations that will turn a `SafeData` object back into a normal `str` or `unicode` object and, rather than try to catch them all, which would be very difficult, Django repairs the damage after the filter has completed.

For example, suppose you have a filter that adds the string `xx` to the end of any input. Since this introduces no dangerous HTML characters to the result (aside from any that were already present), you should mark your filter with `is_safe`:

```
@register.filter(is_safe=True)
def add_xx(value):
    return '%sxx' % value
```

When this filter is used in a template where auto-escaping is enabled, Django will escape the output whenever the input is not already marked as "safe".

By default, `is_safe` is `False`, and you can omit it from any filters where it isn't required.

Be careful when deciding if your filter really does leave safe strings as safe. If you're *removing* characters, you might inadvertently leave unbalanced HTML tags or entities in the result. For example, removing a > from the input might turn <a> into <a, which would need to be escaped on output to avoid causing problems. Similarly, removing a semicolon (`;`) can turn `&` into `&amp`, which is no longer a valid entity and thus needs further escaping. Most cases won't be nearly this tricky, but keep an eye out for any problems like that when reviewing your code.

Marking a filter `is_safe` will coerce the filter's return value to a string. If your filter should return a boolean or other non-string value, marking it `is_safe` will probably have unintended consequences (such as converting a boolean False to the string 'False').

2. Alternatively, your filter code can manually take care of any necessary escaping. This is necessary when you're introducing new HTML markup into the result. You want to mark the output as safe from further escaping so that your HTML markup isn't escaped further, so you'll need to handle the input yourself.

   To mark the output as a safe string, use `django.utils.safestring.mark_safe()`.

   Be careful, though. You need to do more than just mark the output as safe. You need to ensure it really *is* safe, and what you do depends on whether auto-escaping is in effect. The idea is to write filters than can operate in templates where auto-escaping is either on or off in order to make things easier for your template authors.

   In order for your filter to know the current auto-escaping state, set the `needs_autoescape` flag to `True` when you register your filter function. (If you don't specify this flag, it defaults to `False`). This flag tells Django that your filter function wants to be passed an extra keyword argument, called `autoescape`, that is `True` if auto-escaping is in effect and `False` otherwise.

   For example, let's write a filter that emphasizes the first character of a string:

   ```
   from django.utils.html import conditional_escape
   from django.utils.safestring import mark_safe

   @register.filter(needs_autoescape=True)
   def initial_letter_filter(text, autoescape=None):
       first, other = text[0], text[1:]
       if autoescape:
           esc = conditional_escape
       else:
           esc = lambda x: x
       result = '<strong>%s</strong>%s' % (esc(first), esc(other))
       return mark_safe(result)
   ```

   The `needs_autoescape` flag and the `autoescape` keyword argument mean that our function will know whether automatic escaping is in effect when the filter is called. We use `autoescape` to decide whether the input data needs to be passed through `django.utils.html.conditional_escape` or not. (In the latter

case, we just use the identity function as the "escape" function.) The `conditional_escape()` function is like `escape()` except it only escapes input that is **not** a `SafeData` instance. If a `SafeData` instance is passed to `conditional_escape()`, the data is returned unchanged.

Finally, in the above example, we remember to mark the result as safe so that our HTML is inserted directly into the template without further escaping.

There's no need to worry about the `is_safe` flag in this case (although including it wouldn't hurt anything). Whenever you manually handle the auto-escaping issues and return a safe string, the `is_safe` flag won't change anything either way.

Changed in version 1.4: *Please see the release notes* `is_safe` and `needs_autoescape` used to be attributes of the filter function; this syntax is deprecated.

```
@register.filter
def myfilter(value):
    return value
myfilter.is_safe = True

@register.filter
def initial_letter_filter(text, autoescape=None):
    # ...
    return mark_safe(result)
initial_letter_filter.needs_autoescape = True
```

Filters and time zones

New in version 1.4: *Please see the release notes* If you write a custom filter that operates on `datetime` objects, you'll usually register it with the `expects_localtime` flag set to `True`:

```
@register.filter(expects_localtime=True)
def businesshours(value):
    try:
        return 9 <= value.hour < 17
    except AttributeError:
        return ''
```

When this flag is set, if the first argument to your filter is a time zone aware datetime, Django will convert it to the current time zone before passing it to your filter when appropriate, according to *rules for time zones conversions in templates*.

4.5.3 Writing custom template tags

Tags are more complex than filters, because tags can do anything.

A quick overview

Above, this document explained that the template system works in a two-step process: compiling and rendering. To define a custom template tag, you specify how the compilation works and how the rendering works.

When Django compiles a template, it splits the raw template text into ''nodes''. Each node is an instance of `django.template.Node` and has a `render()` method. A compiled template is, simply, a list of `Node` objects. When you call `render()` on a compiled template object, the template calls `render()` on each `Node` in its node list, with the given context. The results are all concatenated together to form the output of the template.

Thus, to define a custom template tag, you specify how the raw template tag is converted into a `Node` (the compilation function), and what the node's `render()` method does.

Writing the compilation function

For each template tag the template parser encounters, it calls a Python function with the tag contents and the parser object itself. This function is responsible for returning a `Node` instance based on the contents of the tag.

For example, let's write a template tag, `{% current_time %}`, that displays the current date/time, formatted according to a parameter given in the tag, in `strftime()` syntax. It's a good idea to decide the tag syntax before anything else. In our case, let's say the tag should be used like this:

```
<p>The time is {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

The parser for this function should grab the parameter and create a `Node` object:

```
from django import template
def do_current_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError("%r tag requires a single argument" % token.contents.split
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError("%r tag's argument should be in quotes" % tag_name)
    return CurrentTimeNode(format_string[1:-1])
```

Notes:

- `parser` is the template parser object. We don't need it in this example.
- `token.contents` is a string of the raw contents of the tag. In our example, it's `'current_time "%Y-%m-%d %I:%M %p"'`.
- The `token.split_contents()` method separates the arguments on spaces while keeping quoted strings together. The more straightforward `token.contents.split()` wouldn't be as robust, as it would naively split on *all* spaces, including those within quoted strings. It's a good idea to always use `token.split_contents()`.
- This function is responsible for raising `django.template.TemplateSyntaxError`, with helpful messages, for any syntax error.
- The `TemplateSyntaxError` exceptions use the `tag_name` variable. Don't hard-code the tag's name in your error messages, because that couples the tag's name to your function. `token.contents.split()[0]` will ''always'' be the name of your tag – even when the tag has no arguments.
- The function returns a `CurrentTimeNode` with everything the node needs to know about this tag. In this case, it just passes the argument – `"%Y-%m-%d %I:%M %p"`. The leading and trailing quotes from the template tag are removed in `format_string[1:-1]`.
- The parsing is very low-level. The Django developers have experimented with writing small frameworks on top of this parsing system, using techniques such as EBNF grammars, but those experiments made the template engine too slow. It's low-level because that's fastest.

Writing the renderer

The second step in writing custom tags is to define a `Node` subclass that has a `render()` method.

Continuing the above example, we need to define `CurrentTimeNode`:

```
from django import template
import datetime
class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string
    def render(self, context):
        return datetime.datetime.now().strftime(self.format_string)
```

Notes:

- `__init__()` gets the `format_string` from `do_current_time()`. Always pass any options/parameters/arguments to a `Node` via its `__init__()`.
- The `render()` method is where the work actually happens.
- `render()` should never raise `TemplateSyntaxError` or any other exception. It should fail silently, just as template filters should.

Ultimately, this decoupling of compilation and rendering results in an efficient template system, because a template can render multiple contexts without having to be parsed multiple times.

Auto-escaping considerations

The output from template tags is **not** automatically run through the auto-escaping filters. However, there are still a couple of things you should keep in mind when writing a template tag.

If the `render()` function of your template stores the result in a context variable (rather than returning the result in a string), it should take care to call `mark_safe()` if appropriate. When the variable is ultimately rendered, it will be affected by the auto-escape setting in effect at the time, so content that should be safe from further escaping needs to be marked as such.

Also, if your template tag creates a new context for performing some sub-rendering, set the auto-escape attribute to the current context's value. The `__init__` method for the `Context` class takes a parameter called `autoescape` that you can use for this purpose. For example:

```
def render(self, context):
    # ...
    new_context = Context({'var': obj}, autoescape=context.autoescape)
    # ... Do something with new_context ...
```

This is not a very common situation, but it's useful if you're rendering a template yourself. For example:

```
def render(self, context):
    t = template.loader.get_template('small_fragment.html')
    return t.render(Context({'var': obj}, autoescape=context.autoescape))
```

If we had neglected to pass in the current `context.autoescape` value to our new `Context` in this example, the results would have *always* been automatically escaped, which may not be the desired behavior if the template tag is used inside a `{% autoescape off %}` block.

Thread-safety considerations

Once a node is parsed, its `render` method may be called any number of times. Since Django is sometimes run in multi-threaded environments, a single node may be simultaneously rendering with different contexts in response to two separate requests. Therefore, it's important to make sure your template tags are thread safe.

To make sure your template tags are thread safe, you should never store state information on the node itself. For example, Django provides a builtin `cycle` template tag that cycles among a list of given strings each time it's rendered:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}>
        ...
    </tr>
{% endfor %}
```

A naive implementation of `CycleNode` might look something like this:

```
class CycleNode(Node):
    def __init__(self, cyclevars):
        self.cycle_iter = itertools.cycle(cyclevars)
    def render(self, context):
        return next(self.cycle_iter)
```

But, suppose we have two templates rendering the template snippet from above at the same time:

1. Thread 1 performs its first loop iteration, `CycleNode.render()` returns 'row1'
2. Thread 2 performs its first loop iteration, `CycleNode.render()` returns 'row2'
3. Thread 1 performs its second loop iteration, `CycleNode.render()` returns 'row1'
4. Thread 2 performs its second loop iteration, `CycleNode.render()` returns 'row2'

The CycleNode is iterating, but it's iterating globally. As far as Thread 1 and Thread 2 are concerned, it's always returning the same value. This is obviously not what we want!

To address this problem, Django provides a `render_context` that's associated with the `context` of the template that is currently being rendered. The `render_context` behaves like a Python dictionary, and should be used to store `Node` state between invocations of the `render` method.

Let's refactor our `CycleNode` implementation to use the `render_context`:

```
class CycleNode(Node):
    def __init__(self, cyclevars):
        self.cyclevars = cyclevars
    def render(self, context):
        if self not in context.render_context:
            context.render_context[self] = itertools.cycle(self.cyclevars)
        cycle_iter = context.render_context[self]
        return next(cycle_iter)
```

Note that it's perfectly safe to store global information that will not change throughout the life of the `Node` as an attribute. In the case of `CycleNode`, the `cyclevars` argument doesn't change after the `Node` is instantiated, so we don't need to put it in the `render_context`. But state information that is specific to the template that is currently being rendered, like the current iteration of the `CycleNode`, should be stored in the `render_context`.

Note: Notice how we used `self` to scope the `CycleNode` specific information within the `render_context`. There may be multiple `CycleNodes` in a given template, so we need to be careful not to clobber another node's state information. The easiest way to do this is to always use `self` as the key into `render_context`. If you're keeping track of several state variables, make `render_context[self]` a dictionary.

Registering the tag

Finally, register the tag with your module's `Library` instance, as explained in "Writing custom template filters" above. Example:

```
register.tag('current_time', do_current_time)
```

The `tag()` method takes two arguments:

1. The name of the template tag – a string. If this is left out, the name of the compilation function will be used.
2. The compilation function – a Python function (not the name of the function as a string).

As with filter registration, it is also possible to use this as a decorator:

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    ...

@register.tag
def shout(parser, token):
    ...
```

If you leave off the `name` argument, as in the second example above, Django will use the function's name as the tag name.

Passing template variables to the tag

Although you can pass any number of arguments to a template tag using `token.split_contents()`, the arguments are all unpacked as string literals. A little more work is required in order to pass dynamic content (a template variable) to a template tag as an argument.

While the previous examples have formatted the current time into a string and returned the string, suppose you wanted to pass in a `DateTimeField` from an object and have the template tag format that date-time:

```
<p>This post was last updated at {% format_time blog_entry.date_updated "%Y-%m-%d %I:%M %p" %}.</p>
```

Initially, `token.split_contents()` will return three values:

1. The tag name `format_time`.
2. The string `"blog_entry.date_updated"` (without the surrounding quotes).
3. The formatting string `"%Y-%m-%d %I:%M %p"`. The return value from `split_contents()` will include the leading and trailing quotes for string literals like this.

Now your tag should begin to look like this:

```
from django import template
def do_format_time(parser, token):
    try:
        # split_contents() knows not to split quoted strings.
        tag_name, date_to_be_formatted, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError("%r tag requires exactly two arguments" % token.contents.s
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError("%r tag's argument should be in quotes" % tag_name)
    return FormatTimeNode(date_to_be_formatted, format_string[1:-1])
```

You also have to change the renderer to retrieve the actual contents of the `date_updated` property of the `blog_entry` object. This can be accomplished by using the `Variable()` class in `django.template`.

To use the `Variable` class, simply instantiate it with the name of the variable to be resolved, and then call `variable.resolve(context)`. So, for example:

```
class FormatTimeNode(template.Node):
    def __init__(self, date_to_be_formatted, format_string):
        self.date_to_be_formatted = template.Variable(date_to_be_formatted)
        self.format_string = format_string

    def render(self, context):
        try:
            actual_date = self.date_to_be_formatted.resolve(context)
            return actual_date.strftime(self.format_string)
        except template.VariableDoesNotExist:
            return ''
```

Variable resolution will throw a `VariableDoesNotExist` exception if it cannot resolve the string passed to it in the current context of the page.

Simple tags

Many template tags take a number of arguments – strings or template variables – and return a string after doing some processing based solely on the input arguments and some external information. For example, the `current_time` tag we wrote above is of this variety: we give it a format string, it returns the time as a string.

To ease the creation of these types of tags, Django provides a helper function, `simple_tag`. This function, which is a method of `django.template.Library`, takes a function that accepts any number of arguments, wraps it in a `render` function and the other necessary bits mentioned above and registers it with the template system.

Our earlier `current_time` function could thus be written like this:

```
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

register.simple_tag(current_time)
```

The decorator syntax also works:

```
@register.simple_tag
def current_time(format_string):
    ...
```

A few things to note about the `simple_tag` helper function:

- Checking for the required number of arguments, etc., has already been done by the time our function is called, so we don't need to do that.
- The quotes around the argument (if any) have already been stripped away, so we just receive a plain string.
- If the argument was a template variable, our function is passed the current value of the variable, not the variable itself.

New in version 1.3: *Please see the release notes* If your template tag needs to access the current context, you can use the `takes_context` argument when registering your tag:

```
# The first argument *must* be called "context" here.
def current_time(context, format_string):
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)

register.simple_tag(takes_context=True)(current_time)
```

Or, using decorator syntax:

```
@register.simple_tag(takes_context=True)
def current_time(context, format_string):
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)
```

For more information on how the `takes_context` option works, see the section on *inclusion tags*. New in version 1.4: *Please see the release notes* If you need to rename your tag, you can provide a custom name for it:

```
register.simple_tag(lambda x: x - 1, name='minusone')

@register.simple_tag(name='minustwo')
def some_function(value):
    return value - 2
```

New in version 1.4: *Please see the release notes* `simple_tag` functions may accept any number of positional or keyword arguments. For example:

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then in the template any number of arguments, separated by spaces, may be passed to the template tag. Like in Python, the values for keyword arguments are set using the equal sign ("=") and must be provided after the positional arguments. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

Inclusion tags

Another common type of template tag is the type that displays some data by rendering *another* template. For example, Django's admin interface uses custom template tags to display the buttons along the bottom of the "add/change" form pages. Those buttons always look the same, but the link targets change depending on the object being edited – so they're a perfect case for using a small template that is filled with details from the current object. (In the admin's case, this is the `submit_row` tag.)

These sorts of tags are called "inclusion tags".

Writing inclusion tags is probably best demonstrated by example. Let's write a tag that outputs a list of choices for a given `Poll` object, such as was created in the *tutorials*. We'll use the tag like this:

```
{% show_results poll %}
```

...and the output will be something like this:

```
<ul>
  <li>First choice</li>
  <li>Second choice</li>
  <li>Third choice</li>
</ul>
```

First, define the function that takes the argument and produces a dictionary of data for the result. The important point here is we only need to return a dictionary, not anything more complex. This will be used as a template context for the template fragment. Example:

```
def show_results(poll):
    choices = poll.choice_set.all()
    return {'choices': choices}
```

Next, create the template used to render the tag's output. This template is a fixed feature of the tag: the tag writer specifies it, not the template designer. Following our example, the template is very simple:

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

Now, create and register the inclusion tag by calling the `inclusion_tag()` method on a `Library` object. Following our example, if the above template is in a file called `results.html` in a directory that's searched by the template loader, we'd register the tag like this:

```
# Here, register is a django.template.Library instance, as before
register.inclusion_tag('results.html')(show_results)
```

Changed in version 1.4: *Please see the release notes* As always, decorator syntax works as well, so we could have written:

```
@register.inclusion_tag('results.html')
def show_results(poll):
    ...
```

...when first creating the function.

Sometimes, your inclusion tags might require a large number of arguments, making it a pain for template authors to pass in all the arguments and remember their order. To solve this, Django provides a `takes_context` option for inclusion tags. If you specify `takes_context` in creating a template tag, the tag will have no required arguments, and the underlying Python function will have one argument – the template context as of when the tag was called.

For example, say you're writing an inclusion tag that will always be used in a context that contains `home_link` and `home_title` variables that point back to the main page. Here's what the Python function would look like:

```
# The first argument *must* be called "context" here.
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
# Register the custom tag as an inclusion tag with takes_context=True.
register.inclusion_tag('link.html', takes_context=True)(jump_link)
```

(Note that the first parameter to the function *must* be called `context`.)

In that `register.inclusion_tag()` line, we specified `takes_context=True` and the name of the template. Here's what the template `link.html` might look like:

```
Jump directly to <a href="{{ link }}">{{ title }}</a>.
```

Then, any time you want to use that custom tag, load its library and call it without any arguments, like so:

```
{% jump_link %}
```

Note that when you're using `takes_context=True`, there's no need to pass arguments to the template tag. It automatically gets access to the context.

The `takes_context` parameter defaults to `False`. When it's set to `True`, the tag is passed the context object, as in this example. That's the only difference between this case and the previous `inclusion_tag` example. New in version 1.4: *Please see the release notes* `inclusion_tag` functions may accept any number of positional or keyword arguments. For example:

```
@register.inclusion_tag('my_template.html')
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then in the template any number of arguments, separated by spaces, may be passed to the template tag. Like in Python, the values for keyword arguments are set using the equal sign ("=") and must be provided after the positional arguments. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

Setting a variable in the context

The above examples simply output a value. Generally, it's more flexible if your template tags set template variables instead of outputting values. That way, template authors can reuse the values that your template tags create.

To set a variable in the context, just use dictionary assignment on the context object in the `render()` method. Here's an updated version of `CurrentTimeNode` that sets a template variable `current_time` instead of outputting it:

```
class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string
    def render(self, context):
        context['current_time'] = datetime.datetime.now().strftime(self.format_string)
        return ''
```

Note that `render()` returns the empty string. `render()` should always return string output. If all the template tag does is set a variable, `render()` should return the empty string.

Here's how you'd use this new version of the tag:

```
{% current_time "%Y-%M-%d %I:%M %p" %}<p>The time is {{ current_time }}.</p>
```

Variable scope in context

Any variable set in the context will only be available in the same `block` of the template in which it was assigned. This behavior is intentional; it provides a scope for variables so that they don't conflict with context in other blocks.

But, there's a problem with `CurrentTimeNode2`: The variable name `current_time` is hard-coded. This means you'll need to make sure your template doesn't use `{{ current_time }}` anywhere else, because the `{% current_time %}` will blindly overwrite that variable's value. A cleaner solution is to make the template tag specify the name of the output variable, like so:

```
{% current_time "%Y-%M-%d %I:%M %p" as my_current_time %}
<p>The current time is {{ my_current_time }}.</p>
```

To do that, you'll need to refactor both the compilation function and `Node` class, like so:

```
class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
```

```
        self.format_string = format_string
        self.var_name = var_name
    def render(self, context):
        context[self.var_name] = datetime.datetime.now().strftime(self.format_string)
        return ''

import re
def do_current_time(parser, token):
    # This version uses a regular expression to parse tag contents.
    try:
        # Splitting by None == splitting by spaces.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        raise template.TemplateSyntaxError("%r tag requires arguments" % token.contents.split()[0])
    m = re.search(r'(.*?) as (\w+)', arg)
    if not m:
        raise template.TemplateSyntaxError("%r tag had invalid arguments" % tag_name)
    format_string, var_name = m.groups()
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError("%r tag's argument should be in quotes" % tag_name)
    return CurrentTimeNode3(format_string[1:-1], var_name)
```

The difference here is that `do_current_time()` grabs the format string and the variable name, passing both to `CurrentTimeNode3`.

Finally, if you only need to have a simple syntax for your custom context-updating template tag, you might want to consider using an *assignment tag*.

Assignment tags

New in version 1.4: *Please see the release notes* To ease the creation of tags setting a variable in the context, Django provides a helper function, `assignment_tag`. This function works the same way as *simple_tag*, except that it stores the tag's result in a specified context variable instead of directly outputting it.

Our earlier `current_time` function could thus be written like this:

```
def get_current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

register.assignment_tag(get_current_time)
```

The decorator syntax also works:

```
@register.assignment_tag
def get_current_time(format_string):
    ...
```

You may then store the result in a template variable using the `as` argument followed by the variable name, and output it yourself where you see fit:

```
{% get_current_time "%Y-%m-%d %I:%M %p" as the_time %}
<p>The time is {{ the_time }}.</p>
```

If your template tag needs to access the current context, you can use the `takes_context` argument when registering your tag:

```
# The first argument *must* be called "context" here.
def get_current_time(context, format_string):
```

```
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)

register.assignment_tag(takes_context=True)(get_current_time)
```

Or, using decorator syntax:

```
@register.assignment_tag(takes_context=True)
def get_current_time(context, format_string):
    timezone = context['timezone']
    return your_get_current_time_method(timezone, format_string)
```

For more information on how the `takes_context` option works, see the section on *inclusion tags*.

`assignment_tag` functions may accept any number of positional or keyword arguments. For example:

```
@register.assignment_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then in the template any number of arguments, separated by spaces, may be passed to the template tag. Like in Python, the values for keyword arguments are set using the equal sign ("=") and must be provided after the positional arguments. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile as the_result %}
```

Parsing until another block tag

Template tags can work in tandem. For instance, the standard `{% comment %}` tag hides everything until `{% endcomment %}`. To create a template tag such as this, use `parser.parse()` in your compilation function.

Here's how a simplified `{% comment %}` tag might be implemented:

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

Note: The actual implementation of `{% comment %}` is slightly different in that it allows broken template tags to appear between `{% comment %}` and `{% endcomment %}`. It does so by calling `parser.skip_past('endcomment')` instead of `parser.parse(('endcomment',))` followed by `parser.delete_first_token()`, thus avoiding the generation of a node list.

`parser.parse()` takes a tuple of names of block tags ''to parse until''. It returns an instance of `django.template.NodeList`, which is a list of all `Node` objects that the parser encountered ''before" it encountered any of the tags named in the tuple.

In `"nodelist = parser.parse(('endcomment',))"` in the above example, `nodelist` is a list of all nodes between the `{% comment %}` and `{% endcomment %}`, not counting `{% comment %}` and `{% endcomment %}` themselves.

After `parser.parse()` is called, the parser hasn't yet "consumed" the `{% endcomment %}` tag, so the code needs to explicitly call `parser.delete_first_token()`.

`CommentNode.render()` simply returns an empty string. Anything between `{% comment %}` and `{% endcomment %}` is ignored.

Parsing until another block tag, and saving contents

In the previous example, `do_comment()` discarded everything between `{% comment %}` and `{% endcomment %}`. Instead of doing that, it's possible to do something with the code between block tags.

For example, here's a custom template tag, `{% upper %}`, that capitalizes everything between itself and `{% endupper %}`.

Usage:

```
{% upper %}This will appear in uppercase, {{ your_name }}.{% endupper %}
```

As in the previous example, we'll use `parser.parse()`. But this time, we pass the resulting `nodelist` to the `Node`:

```
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):
    def __init__(self, nodelist):
        self.nodelist = nodelist
    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

The only new concept here is the `self.nodelist.render(context)` in `UpperNode.render()`.

For more examples of complex rendering, see the source code for `{% if %}`, `{% for %}`, `{% ifequal %}` or `{% ifchanged %}`. They live in `django/template/defaulttags.py`.

4.6 Writing a custom storage system

If you need to provide custom file storage – a common example is storing files on some remote system – you can do so by defining a custom storage class. You'll need to follow these steps:

1. Your custom storage system must be a subclass of `django.core.files.storage.Storage`:

   ```
   from django.core.files.storage import Storage

   class MyStorage(Storage):
       ...
   ```

2. Django must be able to instantiate your storage system without any arguments. This means that any settings should be taken from `django.conf.settings`:

   ```
   from django.conf import settings
   from django.core.files.storage import Storage

   class MyStorage(Storage):
       def __init__(self, option=None):
   ```

```
    if not option:
        option = settings.CUSTOM_STORAGE_OPTIONS
...
```

3. Your storage class must implement the `_open()` and `_save()` methods, along with any other methods appropriate to your storage class. See below for more on these methods.

   In addition, if your class provides local file storage, it must override the `path()` method.

Your custom storage system may override any of the storage methods explained in *File storage API*, but you **must** implement the following methods:

- `Storage.delete()`
- `Storage.exists()`
- `Storage.listdir()`
- `Storage.size()`
- `Storage.url()`

You'll also usually want to use hooks specifically designed for custom storage objects. These are:

4.6.1 `_open(name, mode='rb')`

Required.

Called by `Storage.open()`, this is the actual mechanism the storage class uses to open the file. This must return a `File` object, though in most cases, you'll want to return some subclass here that implements logic specific to the backend storage system.

4.6.2 `_save(name, content)`

Called by `Storage.save()`. The `name` will already have gone through `get_valid_name()` and `get_available_name()`, and the `content` will be a `File` object itself.

Should return the actual name of name of the file saved (usually the `name` passed in, but if the storage needs to change the file name return the new name instead).

4.6.3 `get_valid_name(name)`

Returns a filename suitable for use with the underlying storage system. The `name` argument passed to this method is the original filename sent to the server, after having any path information removed. Override this to customize how non-standard characters are converted to safe filenames.

The code provided on `Storage` retains only alpha-numeric characters, periods and underscores from the original filename, removing everything else.

4.6.4 `get_available_name(name)`

Returns a filename that is available in the storage mechanism, possibly taking the provided filename into account. The `name` argument passed to this method will have already cleaned to a filename valid for the storage system, according to the `get_valid_name()` method described above.

The code provided on `Storage` simply appends `"_1"`, `"_2"`, etc. to the filename until it finds one that's available in the destination directory.

4.7 Deploying Django

Django's chock-full of shortcuts to make Web developer's lives easier, but all those tools are of no use if you can't easily deploy your sites. Since Django's inception, ease of deployment has been a major goal. There's a number of good ways to easily deploy Django:

4.7.1 How to deploy with WSGI

Django's primary deployment platform is WSGI, the Python standard for web servers and applications.

Django's `startproject` management command sets up a simple default WSGI configuration for you, which you can tweak as needed for your project, and direct any WSGI-compliant webserver to use. Django includes getting-started documentation for the following WSGI servers:

How to use Django with Apache and mod_wsgi

Deploying Django with Apache and mod_wsgi is a tried and tested way to get Django into production.

mod_wsgi is an Apache module which can host any Python WSGI application, including Django. Django will work with any version of Apache which supports mod_wsgi.

The official mod_wsgi documentation is fantastic; it's your source for all the details about how to use mod_wsgi. You'll probably want to start with the installation and configuration documentation.

Basic configuration

Once you've got mod_wsgi installed and activated, edit your Apache server's `httpd.conf` file and add:

```
WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py
WSGIPythonPath /path/to/mysite.com

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Order deny,allow
Allow from all
</Files>
</Directory>
```

The first bit in the `WSGIScriptAlias` line is the base URL path you want to serve your application at (`/` indicates the root url), and the second is the location of a "WSGI file" – see below – on your system, usually inside of your project package (`mysite` in this example). This tells Apache to serve any request below the given URL using the WSGI application defined in that file.

The `WSGIPythonPath` line ensures that your project package is available for import on the Python path; in other words, that `import mysite` works.

The `<Directory>` piece just ensures that Apache can access your `wsgi.py` file.

Next we'll need to ensure this `wsgi.py` with a WSGI application object exists. As of Django version 1.4, `startproject` will have created one for you; otherwise, you'll need to create it. See the *WSGI overview documentation* for the default contents you should put in this file, and what else you can add to it.

Using a virtualenv

If you install your project's Python dependencies inside a virtualenv, you'll need to add the path to this virtualenv's `site-packages` directory to your Python path as well. To do this, you can add another line to your Apache configuration:

```
WSGIPythonPath /path/to/your/venv/lib/python2.X/site-packages
```

Make sure you give the correct path to your virtualenv, and replace `python2.X` with the correct Python version (e.g. `python2.7`).

Using mod_wsgi daemon mode

"Daemon mode" is the recommended mode for running mod_wsgi (on non-Windows platforms). See the official mod_wsgi documentation for details on setting up daemon mode. The only change required to the above configuration if you use daemon mode is that you can't use `WSGIPythonPath`; instead you should use the `python-path` option to `WSGIDaemonProcess`, for example:

```
WSGIDaemonProcess example.com python-path=/path/to/mysite.com:/path/to/venv/lib/python2.7/site-packag
```

Serving files

Django doesn't serve files itself; it leaves that job to whichever Web server you choose.

We recommend using a separate Web server – i.e., one that's not also running Django – for serving media. Here are some good choices:

- lighttpd
- Nginx
- TUX
- A stripped-down version of Apache
- Cherokee

If, however, you have no option but to serve media files on the same Apache `VirtualHost` as Django, you can set up Apache to serve some URLs as static media, and others using the mod_wsgi interface to Django.

This example sets up Django at the site root, but explicitly serves `robots.txt`, `favicon.ico`, any CSS file, and anything in the `/static/` and `/media/` URL space as a static file. All other URLs will be served using mod_wsgi:

```
Alias /robots.txt /path/to/mysite.com/static/robots.txt
Alias /favicon.ico /path/to/mysite.com/static/favicon.ico

AliasMatch ^/([^/]*\.css) /path/to/mysite.com/static/styles/$1

Alias /media/ /path/to/mysite.com/media/
Alias /static/ /path/to/mysite.com/static/

<Directory /path/to/mysite.com/static>
Order deny,allow
Allow from all
</Directory>

<Directory /path/to/mysite.com/media>
Order deny,allow
```

```
Allow from all
</Directory>

WSGIScriptAlias / /path/to/mysite.com/mysite/wsgi.py

<Directory /path/to/mysite.com/mysite>
<Files wsgi.py>
Order allow,deny
Allow from all
</Files>
</Directory>
```

Serving the admin files

Note that the Django development server automatically serves the static files of the admin app (and any other installed apps), but this is not the case when you use any other server arrangement. You're responsible for setting up Apache, or whichever media server you're using, to serve the admin files.

The admin files live in (`django/contrib/admin/static/admin`) of the Django distribution.

We **strongly** recommend using `django.contrib.staticfiles` to handle the admin files (along with a Web server as outlined in the previous section; this means using the `collectstatic` management command to collect the static files in `STATIC_ROOT`, and then configuring your Web server to serve `STATIC_ROOT` at `STATIC_URL`), but here are three other approaches:

1. Create a symbolic link to the admin static files from within your document root (this may require `+FollowSymLinks` in your Apache configuration).
2. Use an `Alias` directive, as demonstrated above, to alias the appropriate URL (probably `STATIC_URL` + *admin/*) to the actual location of the admin files.
3. Copy the admin static files so that they live within your Apache document root.

If you get a UnicodeEncodeError

If you're taking advantage of the internationalization features of Django (see *Internationalization and localization*) and you intend to allow users to upload files, you must ensure that the environment used to start Apache is configured to accept non-ASCII file names. If your environment is not correctly configured, you will trigger `UnicodeEncodeError` exceptions when calling functions like `os.path()` on filenames that contain non-ASCII characters.

To avoid these problems, the environment used to start Apache should contain settings analogous to the following:

```
export LANG='en_US.UTF-8'
export LC_ALL='en_US.UTF-8'
```

Consult the documentation for your operating system for the appropriate syntax and location to put these configuration items; `/etc/apache2/envvars` is a common location on Unix platforms. Once you have added these statements to your environment, restart Apache.

How to use Django with Gunicorn

Gunicorn ('Green Unicorn') is a pure-Python WSGI server for UNIX. It has no dependencies and is easy to install and use.

There are two ways to use Gunicorn with Django. One is to have Gunicorn treat Django as just another WSGI application. The second is to use Gunicorn's special integration with Django.

Installing Gunicorn

Installing gunicorn is as easy as `sudo pip install gunicorn`. For more details, see the gunicorn documentation.

Running Django in Gunicorn as a generic WSGI application

When Gunicorn is installed, a `gunicorn` command is available which starts the Gunicorn server process. At its simplest, gunicorn just needs to be called with a the location of a WSGI application object.:

```
gunicorn [OPTIONS] APP_MODULE
```

Where `APP_MODULE` is of the pattern `MODULE_NAME:VARIABLE_NAME`. The module name should be a full dotted path. The variable name refers to a WSGI callable that should be found in the specified module.

So for a typical Django project, invoking gunicorn would look like:

```
gunicorn myproject.wsgi:application
```

(This requires that your project be on the Python path; the simplest way to ensure that is to run this command from the same directory as your `manage.py` file.)

Using Gunicorn's Django integration

To use Gunicorn's built-in Django integration, first add `"gunicorn"` to `INSTALLED_APPS`. Then run `python manage.py run_gunicorn`.

This provides a few Django-specific niceties:

- sets the gunicorn process name to be that of the project
- validates installed models
- allows an `--adminmedia` option for passing in the location of the admin media files.

See Gunicorn's deployment documentation for additional tips on starting and maintaining the Gunicorn server.

How to use Django with uWSGI

uWSGI is a fast, self-healing and developer/sysadmin-friendly application container server coded in pure C.

Prerequisite: uWSGI

The uWSGI wiki describes several installation procedures. Using pip, the Python package manager, you can install any uWSGI version with a single command. For example:

```
# Install current stable version.
$ sudo pip install uwsgi

# Or install LTS (long term support).
$ sudo pip install http://projects.unbit.it/downloads/uwsgi-lts.tar.gz
```

uWSGI model uWSGI operates on a client-server model. Your Web server (e.g., nginx, Apache) communicates with a django-uwsgi "worker" process to serve dynamic content. See uWSGI's background documentation for more detail.

Configuring and starting the uWSGI server for Django uWSGI supports multiple ways to configure the process. See uWSGI's configuration documentation and examples

Here's an example command to start a uWSGI server:

```
uwsgi --chdir=/path/to/your/project \
    --module=mysite.wsgi:application \
    --env DJANGO_SETTINGS_MODULE=mysite.settings \
    --master --pidfile=/tmp/project-master.pid \
    --socket=127.0.0.1:49152 \      # can also be a file
    --processes=5 \                 # number of worker processes
    --uid=1000 --gid=2000 \         # if root, uwsgi can drop privileges
    --harakiri=20 \                 # respawn processes taking more than 20 seconds
    --limit-as=128 \                # limit the project to 128 MB
    --max-requests=5000 \           # respawn processes after serving 5000 requests
    --vacuum \                      # clear environment on exit
    --home=/path/to/virtual/env \   # optional path to a virtualenv
    --daemonize=/var/log/uwsgi/yourproject.log      # background the process
```

This assumes you have a top-level project package named `mysite`, and within it a module `mysite/wsgi.py` that contains a WSGI `application` object. This is the layout you'll have if you ran `django-admin.py startproject mysite` (using your own project name in place of `mysite`) with a recent version of Django. If this file doesn't exist, you'll need to create it. See the *How to deploy with WSGI* documentation for the default contents you should put in this file and what else you can add to it.

The Django-specific options here are:

- `chdir`: The path to the directory that needs to be on Python's import path – i.e., the directory containing the `mysite` package.
- `module`: The WSGI module to use – probably the `mysite.wsgi` module that `startproject` creates.
- `env`: Should probably contain at least `DJANGO_SETTINGS_MODULE`.
- `home`: Optional path to your project virtualenv.

Example ini configuration file:

```
[uwsgi]
chdir=/path/to/your/project
module=mysite.wsgi:application
master=True
pidfile=/tmp/project-master.pid
vacuum=True
max-requests=5000
daemonize=/var/log/uwsgi/yourproject.log
```

Example ini configuration file usage:

```
uwsgi --ini uwsgi.ini
```

See the uWSGI docs on managing the uWSGI process for information on starting, stoping and reloading the uWSGI workers.

The `application` object

One key concept of deploying with WSGI is to specify a central `application` callable object which the webserver uses to communicate with your code. This is commonly specified as an object named `application` in a Python module accessible to the server. Changed in version 1.4: *Please see the release notes* The `startproject` command creates a `projectname/wsgi.py` that contains such an application callable.

Note: Upgrading from a previous release of Django and don't have a `wsgi.py` file in your project? You can simply add one to your project's top-level Python package (probably next to `settings.py` and `urls.py`) with the contents below. If you want `runserver` to also make use of this WSGI file, you can also add `WSGI_APPLICATION = "mysite.wsgi.application"` in your settings (replacing `mysite` with the name of your project).

Initially this file contains:

```
import os

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")

# This application object is used by the development server
# as well as any WSGI server configured to use this file.
from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

The `os.environ.setdefault` line just sets the default settings module to use, if you haven't explicitly set the `DJANGO_SETTINGS_MODULE` environment variable. You'll need to edit this line to replace `mysite` with the name of your project package, so the path to your settings module is correct.

To apply WSGI middleware you can simply wrap the application object in the same file:

```
from helloworld.wsgi import HelloWorldApplication
application = HelloWorldApplication(application)
```

You could also replace the Django WSGI application with a custom WSGI application that later delegates to the Django WSGI application, if you want to combine a Django application with a WSGI application of another framework.

4.7.2 How to use Django with FastCGI, SCGI, or AJP

Although *WSGI* is the preferred deployment platform for Django, many people use shared hosting, on which protocols such as FastCGI, SCGI or AJP are the only viable options.

Note

This document primarily focuses on FastCGI. Other protocols, such as SCGI and AJP, are also supported, through the `flup` Python package. See the Protocols section below for specifics about SCGI and AJP.

Essentially, FastCGI is an efficient way of letting an external application serve pages to a Web server. The Web server delegates the incoming Web requests (via a socket) to FastCGI, which executes the code and passes the response back to the Web server, which, in turn, passes it back to the client's Web browser.

Like WSGI, FastCGI allows code to stay in memory, allowing requests to be served with no startup time. While e.g. *mod_wsgi* can either be configured embedded in the Apache Web server process or as a separate daemon process, a FastCGI process never runs inside the Web server process, always in a separate, persistent process.

Why run code in a separate process?

The traditional `mod_*` arrangements in Apache embed various scripting languages (most notably PHP, Python and Perl) inside the process space of your Web server. Although this lowers startup time – because code doesn't have to be read off disk for every request – it comes at the cost of memory use.

Due to the nature of FastCGI, it's even possible to have processes that run under a different user account than the Web server process. That's a nice security benefit on shared systems, because it means you can secure your code from other users.

Prerequisite: flup

Before you can start using FastCGI with Django, you'll need to install flup, a Python library for dealing with FastCGI. Version 0.5 or newer should work fine.

Starting your FastCGI server

FastCGI operates on a client-server model, and in most cases you'll be starting the FastCGI process on your own. Your Web server (be it Apache, lighttpd, or otherwise) only contacts your Django-FastCGI process when the server needs a dynamic page to be loaded. Because the daemon is already running with the code in memory, it's able to serve the response very quickly.

Note

If you're on a shared hosting system, you'll probably be forced to use Web server-managed FastCGI processes. See the section below on running Django with Web server-managed processes for more information.

A Web server can connect to a FastCGI server in one of two ways: It can use either a Unix domain socket (a "named pipe" on Win32 systems), or it can use a TCP socket. What you choose is a manner of preference; a TCP socket is usually easier due to permissions issues.

To start your server, first change into the directory of your project (wherever your *manage.py* is), and then run the `runfcgi` command:

```
./manage.py runfcgi [options]
```

If you specify `help` as the only option after `runfcgi`, it'll display a list of all the available options.

You'll need to specify either a `socket`, a `protocol` or both `host` and `port`. Then, when you set up your Web server, you'll just need to point it at the host/port or socket you specified when starting the FastCGI server. See the examples, below.

Protocols

Django supports all the protocols that flup does, namely fastcgi, SCGI and AJP1.3 (the Apache JServ Protocol, version 1.3). Select your preferred protocol by using the `protocol=<protocol_name>` option with `./manage.py runfcgi` – where `<protocol_name>` may be one of: `fcgi` (the default), `scgi` or `ajp`. For example:

```
./manage.py runfcgi protocol=scgi
```

Examples

Running a threaded server on a TCP port:

```
./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
```

Running a preforked server on a Unix domain socket:

```
./manage.py runfcgi method=prefork socket=/home/user/mysite.sock pidfile=django.pid
```

Socket security

Django's default umask requires that the webserver and the Django fastcgi process be run with the same group **and** user. For increased security, you can run them under the same group but as different users. If you do this, you will need to set the umask to 0002 using the `umask` argument to `runfcgi`.

Run without daemonizing (backgrounding) the process (good for debugging):

```
./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock maxrequests=1
```

Stopping the FastCGI daemon

If you have the process running in the foreground, it's easy enough to stop it: Simply hitting `Ctrl-C` will stop and quit the FastCGI server. However, when you're dealing with background processes, you'll need to resort to the Unix `kill` command.

If you specify the `pidfile` option to `runfcgi`, you can kill the running FastCGI daemon like this:

```
kill `cat $PIDFILE`
```

...where `$PIDFILE` is the `pidfile` you specified.

To easily restart your FastCGI daemon on Unix, try this small shell script:

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"

cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill `cat -- $PIDFILE`
    rm -f -- $PIDFILE
fi

exec /usr/bin/env - \
  PYTHONPATH="../python:.." \
  ./manage.py runfcgi socket=$SOCKET pidfile=$PIDFILE
```

Apache setup

To use Django with Apache and FastCGI, you'll need Apache installed and configured, with mod_fastcgi installed and enabled. Consult the Apache documentation for instructions.

Once you've got that set up, point Apache at your Django FastCGI instance by editing the `httpd.conf` (Apache configuration) file. You'll need to do two things:

- Use the `FastCGIExternalServer` directive to specify the location of your FastCGI server.

- Use `mod_rewrite` to point URLs at FastCGI as appropriate.

Specifying the location of the FastCGI server

The `FastCGIExternalServer` directive tells Apache how to find your FastCGI server. As the FastCGIExternalServer docs explain, you can specify either a `socket` or a `host`. Here are examples of both:

```
# Connect to FastCGI via a socket / named pipe.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -socket /home/user/mysite.sock

# Connect to FastCGI via a TCP host/port.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 127.0.0.1:3033
```

In either case, the file `/home/user/public_html/mysite.fcgi` doesn't actually have to exist. It's just a URL used by the Web server internally – a hook for signifying which requests at a URL should be handled by FastCGI. (More on this in the next section.)

Using mod_rewrite to point URLs at FastCGI

The second step is telling Apache to use FastCGI for URLs that match a certain pattern. To do this, use the mod_rewrite module and rewrite URLs to `mysite.fcgi` (or whatever you specified in the `FastCGIExternalServer` directive, as explained in the previous section).

In this example, we tell Apache to use FastCGI to handle any request that doesn't represent a file on the filesystem and doesn't start with `/media/`. This is probably the most common case, if you're using Django's admin site:

```
<VirtualHost 12.34.56.78>
  ServerName example.com
  DocumentRoot /home/user/public_html
  Alias /media /home/user/python/django/contrib/admin/media
  RewriteEngine On
  RewriteRule ^/(media.*)$ /$1 [QSA,L,PT]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
</VirtualHost>
```

Django will automatically use the pre-rewrite version of the URL when constructing URLs with the `{% url %}` template tag (and similar methods).

Using mod_fcgid as alternative to mod_fastcgi

Another way to serve applications through FastCGI is by using Apache's mod_fcgid module. Compared to mod_fastcgi mod_fcgid handles FastCGI applications differently in that it manages the spawning of worker processes by itself and doesn't offer something like `FastCGIExternalServer`. This means that the configuration looks slightly different.

In effect, you have to go the way of adding a script handler similar to what is described later on regarding running Django in a *shared-hosting environment*. For further details please refer to the mod_fcgid reference

lighttpd setup

lighttpd is a lightweight Web server commonly used for serving static files. It supports FastCGI natively and, thus, is a good choice for serving both static and dynamic pages, if your site doesn't have any Apache-specific needs.

Make sure `mod_fastcgi` is in your modules list, somewhere after `mod_rewrite` and `mod_access`, but not after `mod_accesslog`. You'll probably want `mod_alias` as well, for serving admin media.

Add the following to your lighttpd config file:

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
            # Use host / port instead of socket for TCP fastcgi
            # "host" => "127.0.0.1",
            # "port" => 3033,
            "socket" => "/home/user/mysite.sock",
            "check-local" => "disable",
        )
    ),
)
alias.url = (
    "/media" => "/home/user/django/contrib/admin/media/",
)

url.rewrite-once = (
    "^(/media.*)$" => "$1",
    "^/favicon\.ico$" => "/media/favicon.ico",
    "^(/.*)$" => "/mysite.fcgi$1",
)
```

Running multiple Django sites on one lighttpd

lighttpd lets you use "conditional configuration" to allow configuration to be customized per host. To specify multiple FastCGI sites, just add a conditional block around your FastCGI config for each site:

```
# If the hostname is 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
    server.document-root = "/foo/site1"
    fastcgi.server = (
        ...
    )
    ...
}

# If the hostname is 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
    server.document-root = "/foo/site2"
    fastcgi.server = (
        ...
    )
    ...
}
```

You can also run multiple Django installations on the same site simply by specifying multiple entries in the `fastcgi.server` directive. Add one FastCGI host for each.

Cherokee setup

Cherokee is a very fast, flexible and easy to configure Web Server. It supports the widespread technologies nowadays: FastCGI, SCGI, PHP, CGI, SSI, TLS and SSL encrypted connections, Virtual hosts, Authentication, on the fly

encoding, Load Balancing, Apache compatible log files, Data Base Balancer, Reverse HTTP Proxy and much more.

The Cherokee project provides a documentation to setting up Django with Cherokee.

Running Django on a shared-hosting provider with Apache

Many shared-hosting providers don't allow you to run your own server daemons or edit the `httpd.conf` file. In these cases, it's still possible to run Django using Web server-spawned processes.

Note

If you're using Web server-spawned processes, as explained in this section, there's no need for you to start the FastCGI server on your own. Apache will spawn a number of processes, scaling as it needs to.

In your Web root directory, add this to a file named `.htaccess`:

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

Then, create a small script that tells Apache how to spawn your FastCGI program. Create a file `mysite.fcgi` and place it in your Web directory, and be sure to make it executable:

```
#!/usr/bin/python
import sys, os

# Add a custom Python path.
sys.path.insert(0, "/home/user/python")

# Switch to the directory of your project. (Optional.)
# os.chdir("/home/user/myproject")

# Set the DJANGO_SETTINGS_MODULE environment variable.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

This works if your server uses mod_fastcgi. If, on the other hand, you are using mod_fcgid the setup is mostly the same except for a slight change in the `.htaccess` file. Instead of adding a fastcgi-script handler, you have to add a fcgid-handler:

```
AddHandler fcgid-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

Restarting the spawned server

If you change any Python code on your site, you'll need to tell FastCGI the code has changed. But there's no need to restart Apache in this case. Rather, just reupload `mysite.fcgi`, or edit the file, so that the timestamp on the file will change. When Apache sees the file has been updated, it will restart your Django application for you.

If you have access to a command shell on a Unix system, you can accomplish this easily by using the `touch` command:

```
touch mysite.fcgi
```

Serving admin media files

Regardless of the server and configuration you eventually decide to use, you will also need to give some thought to how to serve the admin media files. The advice given in the *mod_wsgi* documentation is also applicable in the setups detailed above.

Forcing the URL prefix to a particular value

Because many of these fastcgi-based solutions require rewriting the URL at some point inside the Web server, the path information that Django sees may not resemble the original URL that was passed in. This is a problem if the Django application is being served from under a particular prefix and you want your URLs from the `{% url %}` tag to look like the prefix, rather than the rewritten version, which might contain, for example, `mysite.fcgi`.

Django makes a good attempt to work out what the real script name prefix should be. In particular, if the Web server sets the `SCRIPT_URL` (specific to Apache's mod_rewrite), or `REDIRECT_URL` (set by a few servers, including Apache + mod_rewrite in some situations), Django will work out the original prefix automatically.

In the cases where Django cannot work out the prefix correctly and where you want the original value to be used in URLs, you can set the `FORCE_SCRIPT_NAME` setting in your main `settings` file. This sets the script name uniformly for every URL served via that settings file. Thus you'll need to use different settings files if you want different sets of URLs to have different script names in this case, but that is a rare situation.

As an example of how to use it, if your Django configuration is serving all of the URLs under `'/'` and you wanted to use this setting, you would set `FORCE_SCRIPT_NAME = ''` in your settings file.

If you're new to deploying Django and/or Python, we'd recommend you try *mod_wsgi* first. In most cases it'll be the easiest, fastest, and most stable deployment choice.

See Also:

- Chapter 12 of the Django Book (second edition) discusses deployment and especially scaling in more detail. However, note that this edition was written against Django version 1.1 and has not been updated since *mod_python* was first deprecated, then completely removed in Django 1.5.

4.8 Error reporting

When you're running a public site you should always turn off the `DEBUG` setting. That will make your server run much faster, and will also prevent malicious users from seeing details of your application that can be revealed by the error pages.

However, running with `DEBUG` set to `False` means you'll never see errors generated by your site – everyone will just see your public error pages. You need to keep track of errors that occur in deployed sites, so Django can be configured to create reports with details about those errors.

4.8.1 Email reports

Server errors

When `DEBUG` is `False`, Django will email the users listed in the `ADMINS` setting whenever your code raises an unhandled exception and results in an internal server error (HTTP status code 500). This gives the administrators

immediate notification of any errors. The ADMINS will get a description of the error, a complete Python traceback, and details about the HTTP request that caused the error.

Note: In order to send email, Django requires a few settings telling it how to connect to your mail server. At the very least, you'll need to specify EMAIL_HOST and possibly EMAIL_HOST_USER and EMAIL_HOST_PASSWORD, though other settings may be also required depending on your mail server's configuration. Consult *the Django settings documentation* for a full list of email-related settings.

By default, Django will send email from root@localhost. However, some mail providers reject all email from this address. To use a different sender address, modify the SERVER_EMAIL setting.

To disable this behavior, just remove all entries from the ADMINS setting.

See Also:

New in version 1.3: *Please see the release notes* Server error emails are sent using the logging framework, so you can customize this behavior by *customizing your logging configuration*.

404 errors

Django can also be configured to email errors about broken links (404 "page not found" errors). Django sends emails about 404 errors when:

- DEBUG is False
- SEND_BROKEN_LINK_EMAILS is True
- Your MIDDLEWARE_CLASSES setting includes CommonMiddleware (which it does by default).

If those conditions are met, Django will email the users listed in the MANAGERS setting whenever your code raises a 404 and the request has a referer. (It doesn't bother to email for 404s that don't have a referer – those are usually just people typing in broken URLs or broken Web 'bots).

You can tell Django to stop reporting particular 404s by tweaking the IGNORABLE_404_URLS setting. It should be a tuple of compiled regular expression objects. For example:

```
import re
IGNORABLE_404_URLS = (
    re.compile(r'\.(php|cgi)$'),
    re.compile(r'^/phpmyadmin/'),
)
```

In this example, a 404 to any URL ending with .php or .cgi will *not* be reported. Neither will any URL starting with /phpmyadmin/.

The following example shows how to exclude some conventional URLs that browsers and crawlers often request:

```
import re
IGNORABLE_404_URLS = (
    re.compile(r'^/apple-touch-icon.*\.png$'),
    re.compile(r'^/favicon\.ico$'),
    re.compile(r'^/robots\.txt$'),
)
```

(Note that these are regular expressions, so we put a backslash in front of periods to escape them.)

The best way to disable this behavior is to set SEND_BROKEN_LINK_EMAILS to False.

See Also:

New in version 1.3: *Please see the release notes* 404 errors are logged using the logging framework. By default, these log records are ignored, but you can use them for error reporting by writing a handler and *configuring logging* appropriately.

See Also:

Changed in version 1.4: *Please see the release notes* Previously, two settings were used to control which URLs not to report: `IGNORABLE_404_STARTS` and `IGNORABLE_404_ENDS`. They were replaced by `IGNORABLE_404_URLS`.

4.8.2 Filtering error reports

New in version 1.4: *Please see the release notes*

Filtering sensitive information

Error reports are really helpful for debugging errors, so it is generally useful to record as much relevant information about those errors as possible. For example, by default Django records the full traceback for the exception raised, each traceback frame's local variables, and the `HttpRequest`'s *attributes*.

However, sometimes certain types of information may be too sensitive and thus may not be appropriate to be kept track of, for example a user's password or credit card number. So Django offers a set of function decorators to help you control which information should be filtered out of error reports in a production environment (that is, where `DEBUG` is set to `False`): `sensitive_variables()` and `sensitive_post_parameters()`.

sensitive_variables(**variables*)

If a function (either a view or any regular callback) in your code uses local variables susceptible to contain sensitive information, you may prevent the values of those variables from being included in error reports using the `sensitive_variables` decorator:

```
from django.views.decorators.debug import sensitive_variables

@sensitive_variables('user', 'pw', 'cc')
def process_info(user):
    pw = user.pass_word
    cc = user.credit_card_number
    name = user.name
    ...
```

In the above example, the values for the `user`, `pw` and `cc` variables will be hidden and replaced with stars (**********) in the error reports, whereas the value of the `name` variable will be disclosed.

To systematically hide all local variables of a function from error logs, do not provide any argument to the `sensitive_variables` decorator:

```
@sensitive_variables()
def my_function():
    ...
```

sensitive_post_parameters(**parameters*)

If one of your views receives an `HttpRequest` object with `POST parameters` susceptible to contain sensitive information, you may prevent the values of those parameters from being included in the error reports using the `sensitive_post_parameters` decorator:

```
from django.views.decorators.debug import sensitive_post_parameters

@sensitive_post_parameters('pass_word', 'credit_card_number')
```

```
def record_user_profile(request):
    UserProfile.create(user=request.user,
                       password=request.POST['pass_word'],
                       credit_card=request.POST['credit_card_number'],
                       name=request.POST['name'])
    ...
```

In the above example, the values for the `pass_word` and `credit_card_number` POST parameters will be hidden and replaced with stars (**********) in the request's representation inside the error reports, whereas the value of the `name` parameter will be disclosed.

To systematically hide all POST parameters of a request in error reports, do not provide any argument to the `sensitive_post_parameters` decorator:

```
@sensitive_post_parameters()
def my_view(request):
    ...
```

Note: Changed in version 1.4: *Please see the release notes* Since version 1.4, all POST parameters are systematically filtered out of error reports for certain `contrib.views.auth` views (`login`, `password_reset_confirm`, `password_change`, and `add_view` and `user_change_password` in the `auth` admin) to prevent the leaking of sensitive information such as user passwords.

Custom error reports

All `sensitive_variables()` and `sensitive_post_parameters()` do is, respectively, annotate the decorated function with the names of sensitive variables and annotate the `HttpRequest` object with the names of sensitive POST parameters, so that this sensitive information can later be filtered out of reports when an error occurs. The actual filtering is done by Django's default error reporter filter: `django.views.debug.SafeExceptionReporterFilter`. This filter uses the decorators' annotations to replace the corresponding values with stars (**********) when the error reports are produced. If you wish to override or customize this default behavior for your entire site, you need to define your own filter class and tell Django to use it via the `DEFAULT_EXCEPTION_REPORTER_FILTER` setting:

```
DEFAULT_EXCEPTION_REPORTER_FILTER = 'path.to.your.CustomExceptionReporterFilter'
```

You may also control in a more granular way which filter to use within any given view by setting the `HttpRequest`'s `exception_reporter_filter` attribute:

```
def my_view(request):
    if request.user.is_authenticated():
        request.exception_reporter_filter = CustomExceptionReporterFilter()
    ...
```

Your custom filter class needs to inherit from `django.views.debug.SafeExceptionReporterFilter` and may override the following methods:

class `django.views.debug.`**`SafeExceptionReporterFilter`**

`SafeExceptionReporterFilter.`**`is_active`**(*self*, *request*)
Returns `True` to activate the filtering operated in the other methods. By default the filter is active if `DEBUG` is `False`.

`SafeExceptionReporterFilter.`**`get_request_repr`**(*self*, *request*)
Returns the representation string of the request object, that is, the value that would be returned by `repr(request)`, except it uses the filtered dictionary of POST parameters as determined by `SafeExceptionReporterFilter.get_post_parameters()`.

`SafeExceptionReporterFilter.`**`get_post_parameters`**(*self*, *request*)
Returns the filtered dictionary of POST parameters. By default it replaces the values of sensitive parameters with stars (**********).

`SafeExceptionReporterFilter.`**`get_traceback_frame_variables`**(*self*, *request*, *tb_frame*)
Returns the filtered dictionary of local variables for the given traceback frame. By default it replaces the values of sensitive variables with stars (**********).

See Also:

You can also set up custom error reporting by writing a custom piece of *exception middleware*. If you do write custom error handling, it's a good idea to emulate Django's built-in error handling and only report/log errors if `DEBUG` is `False`.

4.9 Providing initial data for models

It's sometimes useful to pre-populate your database with hard-coded data when you're first setting up an app. There's a couple of ways you can have Django automatically create this data: you can provide initial data via fixtures, or you can provide initial data as SQL.

In general, using a fixture is a cleaner method since it's database-agnostic, but initial SQL is also quite a bit more flexible.

4.9.1 Providing initial data with fixtures

A fixture is a collection of data that Django knows how to import into a database. The most straightforward way of creating a fixture if you've already got some data is to use the `manage.py dumpdata` command. Or, you can write fixtures by hand; fixtures can be written as XML, YAML, or JSON documents. The *serialization documentation* has more details about each of these supported *serialization formats*.

As an example, though, here's what a fixture for a simple `Person` model might look like in JSON:

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

And here's that same fixture as YAML:

```
- model: myapp.person
  pk: 1
  fields:
```

```
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

You'll store this data in a `fixtures` directory inside your app.

Loading data is easy: just call `manage.py loaddata <fixturename>`, where `<fixturename>` is the name of the fixture file you've created. Each time you run `loaddata`, the data will be read from the fixture and re-loaded into the database. Note this means that if you change one of the rows created by a fixture and then run `loaddata` again, you'll wipe out any changes you've made.

Automatically loading initial data fixtures

If you create a fixture named `initial_data.[xml/yaml/json]`, that fixture will be loaded every time you run `syncdb`. This is extremely convenient, but be careful: remember that the data will be refreshed *every time* you run `syncdb`. So don't use `initial_data` for data you'll want to edit.

Where Django finds fixture files

By default, Django looks in the `fixtures` directory inside each app for fixtures. You can set the `FIXTURE_DIRS` setting to a list of additional directories where Django should look.

When running `manage.py loaddata`, you can also specify an absolute path to a fixture file, which overrides searching the usual directories.

See Also:

Fixtures are also used by the *testing framework* to help set up a consistent test environment.

4.9.2 Providing initial SQL data

Django provides a hook for passing the database arbitrary SQL that's executed just after the CREATE TABLE statements when you run `syncdb`. You can use this hook to populate default records, or you could also create SQL functions, views, triggers, etc.

The hook is simple: Django just looks for a file called `sql/<modelname>.sql`, in your app directory, where `<modelname>` is the model's name in lowercase.

So, if you had a `Person` model in an app called `myapp`, you could add arbitrary SQL to the file `sql/person.sql` inside your `myapp` directory. Here's an example of what the file might contain:

```
INSERT INTO myapp_person (first_name, last_name) VALUES ('John', 'Lennon');
INSERT INTO myapp_person (first_name, last_name) VALUES ('Paul', 'McCartney');
```

Each SQL file, if given, is expected to contain valid SQL statements which will insert the desired data (e.g., properly-formatted `INSERT` statements separated by semicolons).

The SQL files are read by the `sqlcustom` and `sqlall` commands in *manage.py*. Refer to the *manage.py documentation* for more information.

Note that if you have multiple SQL data files, there's no guarantee of the order in which they're executed. The only thing you can assume is that, by the time your custom data files are executed, all the database tables already will have been created.

Initial SQL data and testing

This technique *cannot* be used to provide initial data for testing purposes. Django's test framework flushes the contents of the test database after each test; as a result, any data added using the custom SQL hook will be lost.

If you require data for a test case, you should add it using either a *test fixture*, or programatically add it during the `setUp()` of your test case.

Database-backend-specific SQL data

There's also a hook for backend-specific SQL data. For example, you can have separate initial-data files for PostgreSQL and SQLite. For each app, Django looks for a file called `<appname>/sql/<modelname>.<backend>.sql`, where `<appname>` is your app directory, `<modelname>` is the model's name in lowercase and `<backend>` is the last part of the module name provided for the `ENGINE` in your settings file (e.g., if you have defined a database with an `ENGINE` value of `django.db.backends.sqlite3`, Django will look for `<appname>/sql/<modelname>.sqlite3.sql`).

Backend-specific SQL data is executed before non-backend-specific SQL data. For example, if your app contains the files `sql/person.sql` and `sql/person.sqlite3.sql` and you're installing the app on SQLite, Django will execute the contents of `sql/person.sqlite3.sql` first, then `sql/person.sql`.

4.10 Running Django on Jython

Python 2.6 support

Django 1.5 has dropped support for Python 2.5. Until Jython provides a new version that supports 2.6, Django 1.5 is no more compatible with Jython. Please use Django 1.4 if you want to use Django over Jython.

Jython is an implementation of Python that runs on the Java platform (JVM). Django runs cleanly on Jython version 2.5 or later, which means you can deploy Django on any Java platform.

This document will get you up and running with Django on top of Jython.

4.10.1 Installing Jython

Django works with Jython versions 2.5b3 and higher. Download Jython at http://www.jython.org/.

4.10.2 Creating a servlet container

If you just want to experiment with Django, skip ahead to the next section; Django includes a lightweight Web server you can use for testing, so you won't need to set up anything else until you're ready to deploy Django in production.

If you want to use Django on a production site, use a Java servlet container, such as Apache Tomcat. Full JavaEE applications servers such as GlassFish or JBoss are also OK, if you need the extra features they include.

4.10.3 Installing Django

The next step is to install Django itself. This is exactly the same as installing Django on standard Python, so see *Remove any old versions of Django* and *Install the Django code* for instructions.

4.10.4 Installing Jython platform support libraries

The django-jython project contains database backends and management commands for Django/Jython development. Note that the builtin Django backends won't work on top of Jython.

To install it, follow the installation instructions detailed on the project Web site. Also, read the database backends documentation there.

4.10.5 Differences with Django on Jython

At this point, Django on Jython should behave nearly identically to Django running on standard Python. However, are a few differences to keep in mind:

- Remember to use the `jython` command instead of `python`. The documentation uses `python` for consistency, but if you're using Jython you'll want to mentally replace `python` with `jython` every time it occurs.
- Similarly, you'll need to use the `JYTHONPATH` environment variable instead of `PYTHONPATH`.

4.11 Integrating Django with a legacy database

While Django is best suited for developing new applications, it's quite possible to integrate it into legacy databases. Django includes a couple of utilities to automate as much of this process as possible.

This document assumes you know the Django basics, as covered in the *tutorial*.

Once you've got Django set up, you'll follow this general process to integrate with an existing database.

4.11.1 Give Django your database parameters

You'll need to tell Django what your database connection parameters are, and what the name of the database is. Do that by editing the `DATABASES` setting and assigning values to the following keys for the `'default'` connection:

- `NAME`
- `ENGINE`
- `USER`
- `PASSWORD`
- `HOST`
- `PORT`

4.11.2 Auto-generate the models

Django comes with a utility called `inspectdb` that can create models by introspecting an existing database. You can view the output by running this command:

```
python manage.py inspectdb
```

Save this as a file by using standard Unix output redirection:

```
python manage.py inspectdb > models.py
```

This feature is meant as a shortcut, not as definitive model generation. See the documentation of inspectdb for more information.

Once you've cleaned up your models, name the file models.py and put it in the Python package that holds your app. Then add the app to your INSTALLED_APPS setting.

4.11.3 Install the core Django tables

Next, run the syncdb command to install any extra needed database records such as admin permissions and content types:

```
python manage.py syncdb
```

4.11.4 Test and tweak

Those are the basic steps – from here you'll want to tweak the models Django generated until they work the way you'd like. Try accessing your data via the Django database API, and try editing objects via Django's admin site, and edit the models file accordingly.

4.12 Outputting CSV with Django

This document explains how to output CSV (Comma Separated Values) dynamically using Django views. To do this, you can either use the Python CSV library or the Django template system.

4.12.1 Using the Python CSV library

Python comes with a CSV library, csv. The key to using it with Django is that the csv module's CSV-creation capability acts on file-like objects, and Django's HttpResponse objects are file-like objects.

Here's an example:

```
import csv
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=somefilename.csv'

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo', 'Bar', 'Baz'])
    writer.writerow(['Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"])

    return response
```

The code and comments should be self-explanatory, but a few things deserve a mention:

- The response gets a special MIME type, *text/csv*. This tells browsers that the document is a CSV file, rather than an HTML file. If you leave this off, browsers will probably interpret the output as HTML, which will result in ugly, scary gobbledygook in the browser window.
- The response gets an additional `Content-Disposition` header, which contains the name of the CSV file. This filename is arbitrary; call it whatever you want. It'll be used by browsers in the "Save as..." dialogue, etc.
- Hooking into the CSV-generation API is easy: Just pass `response` as the first argument to `csv.writer`. The `csv.writer` function expects a file-like object, and `HttpResponse` objects fit the bill.
- For each row in your CSV file, call `writer.writerow`, passing it an iterable object such as a list or tuple.
- The CSV module takes care of quoting for you, so you don't have to worry about escaping strings with quotes or commas in them. Just pass `writerow()` your raw strings, and it'll do the right thing.

Handling Unicode

Python's `csv` module does not support Unicode input. Since Django uses Unicode internally this means strings read from sources such as `HttpRequest` are potentially problematic. There are a few options for handling this:

- Manually encode all Unicode objects to a compatible encoding.
- Use the `UnicodeWriter` class provided in the csv module's examples section.
- Use the python-unicodecsv module, which aims to be a drop-in replacement for `csv` that gracefully handles Unicode.

For more information, see the Python documentation of the `csv` module.

4.12.2 Using the template system

Alternatively, you can use the *Django template system* to generate CSV. This is lower-level than using the convenient Python `csv` module, but the solution is presented here for completeness.

The idea here is to pass a list of items to your template, and have the template output the commas in a `for` loop.

Here's an example, which generates the same CSV file as above:

```
from django.http import HttpResponse
from django.template import loader, Context

def some_view(request):
    # Create the HttpResponse object with the appropriate CSV header.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=somefilename.csv'

    # The data is hard-coded here, but you could load it from a database or
    # some other source.
    csv_data = (
        ('First row', 'Foo', 'Bar', 'Baz'),
        ('Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"),
    )

    t = loader.get_template('my_template_name.txt')
    c = Context({
        'data': csv_data,
    })
    response.write(t.render(c))
    return response
```

The only difference between this example and the previous example is that this one uses template loading instead of the CSV module. The rest of the code – such as the `mimetype='text/csv'` – is the same.

Then, create the template `my_template_name.txt`, with this template code:

```
{% for row in data %}"{{ row.0|addslashes }}", "{{ row.1|addslashes }}", "{{ row.2|addslashes }}", "
{% endfor %}
```

This template is quite basic. It just iterates over the given data and displays a line of CSV for each row. It uses the `addslashes` template filter to ensure there aren't any problems with quotes.

4.12.3 Other text-based formats

Notice that there isn't very much specific to CSV here – just the specific output format. You can use either of these techniques to output any text-based format you can dream of. You can also use a similar technique to generate arbitrary binary data; see *Outputting PDFs with Django* for an example.

4.13 Outputting PDFs with Django

This document explains how to output PDF files dynamically using Django views. This is made possible by the excellent, open-source ReportLab Python PDF library.

The advantage of generating PDF files dynamically is that you can create customized PDFs for different purposes – say, for different users or different pieces of content.

For example, Django was used at kusports.com to generate customized, printer-friendly NCAA tournament brackets, as PDF files, for people participating in a March Madness contest.

4.13.1 Install ReportLab

Download and install the ReportLab library from http://www.reportlab.com/software/opensource/rl-toolkit/download/. The user guide (not coincidentally, a PDF file) explains how to install it. Alternatively, you can also install it with `pip`:

```
$ sudo pip install reportlab
```

Test your installation by importing it in the Python interactive interpreter:

```
>>> import reportlab
```

If that command doesn't raise any errors, the installation worked.

4.13.2 Write your view

The key to generating PDFs dynamically with Django is that the ReportLab API acts on file-like objects, and Django's `HttpResponse` objects are file-like objects.

Here's a "Hello World" example:

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
```

```
    response['Content-Disposition'] = 'attachment; filename=somefilename.pdf'

    # Create the PDF object, using the response object as its "file."
    p = canvas.Canvas(response)

    # Draw things on the PDF. Here's where the PDF generation happens.
    # See the ReportLab documentation for the full list of functionality.
    p.drawString(100, 100, "Hello world.")

    # Close the PDF object cleanly, and we're done.
    p.showPage()
    p.save()
    return response
```

The code and comments should be self-explanatory, but a few things deserve a mention:

- The response gets a special MIME type, *application/pdf*. This tells browsers that the document is a PDF file, rather than an HTML file. If you leave this off, browsers will probably interpret the output as HTML, which would result in ugly, scary gobbledygook in the browser window.
- The response gets an additional `Content-Disposition` header, which contains the name of the PDF file. This filename is arbitrary: Call it whatever you want. It'll be used by browsers in the "Save as..." dialogue, etc.
- The `Content-Disposition` header starts with `'attachment; '` in this example. This forces Web browsers to pop-up a dialog box prompting/confirming how to handle the document even if a default is set on the machine. If you leave off `'attachment;'`, browsers will handle the PDF using whatever program/plugin they've been configured to use for PDFs. Here's what that code would look like:

  ```
  response['Content-Disposition'] = 'filename=somefilename.pdf'
  ```

- Hooking into the ReportLab API is easy: Just pass `response` as the first argument to `canvas.Canvas`. The `Canvas` class expects a file-like object, and `HttpResponse` objects fit the bill.
- Note that all subsequent PDF-generation methods are called on the PDF object (in this case, `p`) – not on `response`.
- Finally, it's important to call `showPage()` and `save()` on the PDF file.

Note: ReportLab is not thread-safe. Some of our users have reported odd issues with building PDF-generating Django views that are accessed by many people at the same time.

4.13.3 Complex PDFs

If you're creating a complex PDF document with ReportLab, consider using the `io` library as a temporary holding place for your PDF file. This library provides a file-like object interface that is particularly efficient. Here's the above "Hello World" example rewritten to use `io`:

```
from io import BytesIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Create the HttpResponse object with the appropriate PDF headers.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=somefilename.pdf'

    buffer = BytesIO()
```

```
# Create the PDF object, using the BytesIO object as its "file."
p = canvas.Canvas(buffer)

# Draw things on the PDF. Here's where the PDF generation happens.
# See the ReportLab documentation for the full list of functionality.
p.drawString(100, 100, "Hello world.")

# Close the PDF object cleanly.
p.showPage()
p.save()

# Get the value of the BytesIO buffer and write it to the response.
pdf = buffer.getvalue()
buffer.close()
response.write(pdf)
return response
```

4.13.4 Further resources

- PDFlib is another PDF-generation library that has Python bindings. To use it with Django, just use the same concepts explained in this article.
- Pisa XHTML2PDF is yet another PDF-generation library. Pisa ships with an example of how to integrate Pisa with Django.
- HTMLdoc is a command-line script that can convert HTML to PDF. It doesn't have a Python interface, but you can escape out to the shell using `system` or `popen` and retrieve the output in Python.

4.13.5 Other formats

Notice that there isn't a lot in these examples that's PDF-specific – just the bits using `reportlab`. You can use a similar technique to generate any arbitrary format that you can find a Python library for. Also see *Outputting CSV with Django* for another example and some techniques you can use when generated text-based formats.

4.14 Managing static files

New in version 1.3: *Please see the release notes* Django developers mostly concern themselves with the dynamic parts of web applications – the views and templates that render anew for each request. But web applications have other parts: the static files (images, CSS, Javascript, etc.) that are needed to render a complete web page.

For small projects, this isn't a big deal, because you can just keep the static files somewhere your web server can find it. However, in bigger projects – especially those comprised of multiple apps – dealing with the multiple sets of static files provided by each application starts to get tricky.

That's what `django.contrib.staticfiles` is for: it collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

Note: If you've used the django-staticfiles third-party app before, then `django.contrib.staticfiles` will look very familiar. That's because they're essentially the same code: `django.contrib.staticfiles` started its life as django-staticfiles and was merged into Django 1.3.

If you're upgrading from `django-staticfiles`, please see Upgrading from django-staticfiles, below, for a few minor changes you'll need to make.

4.14.1 Using `django.contrib.staticfiles`

Basic usage

1. Put your static files somewhere that `staticfiles` will find them.

   By default, this means within `static/` subdirectories of apps in your `INSTALLED_APPS`.

   Your project will probably also have static assets that aren't tied to a particular app. The `STATICFILES_DIRS` setting is a tuple of filesystem directories to check when loading static files. It's a search path that is by default empty. See the `STATICFILES_DIRS` docs how to extend this list of additional paths.

   Additionally, see the documentation for the `STATICFILES_FINDERS` setting for details on how `staticfiles` finds your files.

2. Make sure that `django.contrib.staticfiles` is included in your `INSTALLED_APPS`.

   For *local development*, if you are using *runserver* or adding *staticfiles_urlpatterns* to your URLconf, you're done with the setup – your static files will automatically be served at the default (for `newly created` projects) `STATIC_URL` of `/static/`.

3. You'll probably need to refer to these files in your templates. The easiest method is to use the included context processor which allows template code like:

   ```
   <img src="{{ STATIC_URL }}images/hi.jpg" alt="Hi!" />
   ```

   See *Referring to static files in templates* for more details, **including** an alternate method using a template tag.

Deploying static files in a nutshell

When you're ready to move out of local development and deploy your project:

1. Set the `STATIC_URL` setting to the public URL for your static files (in most cases, the default value of `/static/` is just fine).

2. Set the `STATIC_ROOT` setting to point to the filesystem path you'd like your static files collected to when you use the `collectstatic` management command. For example:

   ```
   STATIC_ROOT = "/home/jacob/projects/mysite.com/sitestatic"
   ```

3. Run the `collectstatic` management command:

   ```
   ./manage.py collectstatic
   ```

   This'll churn through your static file storage and copy them into the directory given by `STATIC_ROOT`.

4. Deploy those files by configuring your webserver of choice to serve the files in `STATIC_ROOT` at `STATIC_URL`.

   Serving static files in production covers some common deployment strategies for static files.

Those are the **basics**. For more details on common configuration options, read on; for a detailed reference of the settings, commands, and other bits included with the framework see *the staticfiles reference*.

Note: In previous versions of Django, it was common to place static assets in `MEDIA_ROOT` along with user-uploaded files, and serve them both at `MEDIA_URL`. Part of the purpose of introducing the `staticfiles` app is to make it easier to keep static files separate from user-uploaded files.

For this reason, you need to make your MEDIA_ROOT and MEDIA_URL different from your STATIC_ROOT and STATIC_URL. You will need to arrange for serving of files in MEDIA_ROOT yourself; staticfiles does not deal with user-uploaded files at all. You can, however, use django.views.static.serve() view for serving MEDIA_ROOT in development; see *Serving other directories*.

4.14.2 Referring to static files in templates

At some point, you'll probably need to link to static files in your templates. You could, of course, simply hardcode the path to you assets in the templates:

```
<img src="http://static.example.com/static/myimage.jpg" alt="Sample image" />
```

Of course, there are some serious problems with this: it doesn't work well in development, and it makes it *very* hard to change where you've deployed your static files. If, for example, you wanted to switch to using a content delivery network (CDN), then you'd need to change more or less every single template.

A far better way is to use the value of the STATIC_URL setting directly in your templates. This means that a switch of static files servers only requires changing that single value. Much better!

Django includes multiple built-in ways of using this setting in your templates: a context processor and a template tag.

With a context processor

The included context processor is the easy way. Simply make sure 'django.core.context_processors.static' is in your TEMPLATE_CONTEXT_PROCESSORS. It's there by default, and if you're editing that setting by hand it should look something like:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.debug',
    'django.core.context_processors.i18n',
    'django.core.context_processors.media',
    'django.core.context_processors.static',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages',
)
```

Once that's done, you can refer to STATIC_URL in your templates:

```
<img src="{{ STATIC_URL }}images/hi.jpg" alt="Hi!" />
```

If {{ STATIC_URL }} isn't working in your template, you're probably not using RequestContext when rendering the template.

As a brief refresher, context processors add variables into the contexts of every template. However, context processors require that you use RequestContext when rendering templates. This happens automatically if you're using a *generic view*, but in views written by hand you'll need to explicitly use RequestContext To see how that works, and to read more details, check out *Subclassing Context: RequestContext*.

Another option is the get_static_prefix template tag that is part of Django's core.

With a template tag

The more powerful tool is the static template tag. It builds the URL for the given relative path by using the configured STATICFILES_STORAGE storage.

```
{% load staticfiles %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!"/>
```

It is also able to consume standard context variables, e.g. assuming a `user_stylesheet` variable is passed to the template:

```
{% load staticfiles %}
<link rel="stylesheet" href="{% static user_stylesheet %}" type="text/css" media="screen" />
```

Note: There is also a template tag named `static` in Django's core set of *built in template tags* which has the same argument signature but only uses urlparse.urljoin() with the `STATIC_URL` setting and the given path. This has the disadvantage of not being able to easily switch the storage backend without changing the templates, so in doubt use the `staticfiles static` template tag.

4.14.3 Serving static files in development

The static files tools are mostly designed to help with getting static files successfully deployed into production. This usually means a separate, dedicated static file server, which is a lot of overhead to mess with when developing locally. Thus, the `staticfiles` app ships with a **quick and dirty helper view** that you can use to serve files locally in development.

This view is automatically enabled and will serve your static files at `STATIC_URL` when you use the built-in *runserver* management command.

To enable this view if you are using some other server for local development, you'll add a couple of lines to your URLconf. The first line goes at the top of the file, and the last line at the bottom:

```
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

# ... the rest of your URLconf goes here ...

urlpatterns += staticfiles_urlpatterns()
```

This will inspect your `STATIC_URL` setting and wire up the view to serve static files accordingly. Don't forget to set the `STATICFILES_DIRS` setting appropriately to let `django.contrib.staticfiles` know where to look for files additionally to files in app directories.

> **Warning:** This will only work if `DEBUG` is `True`.
> That's because this view is **grossly inefficient** and probably **insecure**. This is only intended for local development, and should **never be used in production**.
> Additionally, when using `staticfiles_urlpatterns` your `STATIC_URL` setting can't be empty or a full URL, such as `http://static.example.com/`.

For a few more details on how the `staticfiles` can be used during development, see *Static file development view*.

Serving other directories

serve(*request*, *path*, *document_root*, *show_indexes=False*)

There may be files other than your project's static assets that, for convenience, you'd like to have Django serve for you in local development. The `serve()` view can be used to serve any directory you give it. (Again, this view is **not** hardened for production use, and should be used only as a development aid; you should serve these files in production using a real front-end webserver).

The most likely example is user-uploaded content in MEDIA_ROOT. staticfiles is intended for static assets and has no built-in handling for user-uploaded files, but you can have Django serve your MEDIA_ROOT by appending something like this to your URLconf:

```
from django.conf import settings

# ... the rest of your URLconf goes here ...

if settings.DEBUG:
    urlpatterns += patterns('',
        url(r'^media/(?P<path>.*)$', 'django.views.static.serve', {
            'document_root': settings.MEDIA_ROOT,
        }),
   )
```

Note, the snippet assumes your MEDIA_URL has a value of '/media/'. This will call the serve() view, passing in the path from the URLconf and the (required) document_root parameter.

static(*prefix*, *view='django.views.static.serve'*, ***kwargs*)

Since it can become a bit cumbersome to define this URL pattern, Django ships with a small URL helper function static() that takes as parameters the prefix such as MEDIA_URL and a dotted path to a view, such as 'django.views.static.serve'. Any other function parameter will be transparently passed to the view.

An example for serving MEDIA_URL ('/media/') during development:

```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = patterns('',
    # ... the rest of your URLconf goes here ...
) + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Note: This helper function will only be operational in debug mode and if the given prefix is local (e.g. /static/) and not a URL (e.g. http://static.example.com/).

4.14.4 Serving static files in production

The basic outline of putting static files into production is simple: run the collectstatic command when static files change, then arrange for the collected static files directory (STATIC_ROOT) to be moved to the static file server and served.

Of course, as with all deployment tasks, the devil's in the details. Every production setup will be a bit different, so you'll need to adapt the basic outline to fit your needs. Below are a few common patterns that might help.

Serving the app and your static files from the same server

If you want to serve your static files from the same server that's already serving your site, the basic outline gets modified to look something like:

- Push your code up to the deployment server.
- On the server, run collectstatic to copy all the static files into STATIC_ROOT.
- Point your web server at STATIC_ROOT. For example, here's *how to do this under Apache and mod_wsgi*.

You'll probably want to automate this process, especially if you've got multiple web servers. There's any number of ways to do this automation, but one option that many Django developers enjoy is Fabric.

Below, and in the following sections, we'll show off a few example fabfiles (i.e. Fabric scripts) that automate these file deployment options. The syntax of a fabfile is fairly straightforward but won't be covered here; consult Fabric's documentation, for a complete explanation of the syntax..

So, a fabfile to deploy static files to a couple of web servers might look something like:

```
from fabric.api import *

# Hosts to deploy onto
env.hosts = ['www1.example.com', 'www2.example.com']

# Where your project code lives on the server
env.project_root = '/home/www/myproject'

def deploy_static():
    with cd(env.project_root):
        run('./manage.py collectstatic -v0 --noinput')
```

Serving static files from a dedicated server

Most larger Django apps use a separate Web server – i.e., one that's not also running Django – for serving static files. This server often runs a different type of web server – faster but less full-featured. Some good choices are:

- lighttpd
- Nginx
- TUX
- Cherokee
- A stripped-down version of Apache

Configuring these servers is out of scope of this document; check each server's respective documentation for instructions.

Since your static file server won't be running Django, you'll need to modify the deployment strategy to look something like:

- When your static files change, run `collectstatic` locally.
- Push your local `STATIC_ROOT` up to the static file server into the directory that's being served. `rsync` is a good choice for this step since it only needs to transfer the bits of static files that have changed.

Here's how this might look in a fabfile:

```
from fabric.api import *
from fabric.contrib import project

# Where the static files get collected locally
env.local_static_root = '/tmp/static'

# Where the static files should go remotely
env.remote_static_root = '/home/www/static.example.com'

@roles('static')
def deploy_static():
    local('./manage.py collectstatic')
```

```
project.rsync_project(
    remote_dir = env.remote_static_root,
    local_dir = env.local_static_root,
    delete = True
)
```

Serving static files from a cloud service or CDN

Another common tactic is to serve static files from a cloud storage provider like Amazon's S3 and/or a CDN (content delivery network). This lets you ignore the problems of serving static files, and can often make for faster-loading webpages (especially when using a CDN).

When using these services, the basic workflow would look a bit like the above, except that instead of using `rsync` to transfer your static files to the server you'd need to transfer the static files to the storage provider or CDN.

There's any number of ways you might do this, but if the provider has an API a *custom file storage backend* will make the process incredibly simple. If you've written or are using a 3rd party custom storage backend, you can tell `collectstatic` to use it by setting `STATICFILES_STORAGE` to the storage engine.

For example, if you've written an S3 storage backend in `myproject.storage.S3Storage` you could use it with:

```
STATICFILES_STORAGE = 'myproject.storage.S3Storage'
```

Once that's done, all you have to do is run `collectstatic` and your static files would be pushed through your storage package up to S3. If you later needed to switch to a different storage provider, it could be as simple as changing your `STATICFILES_STORAGE` setting.

For details on how you'd write one of these backends, *Writing a custom storage system*.

See Also:

The django-storages project is a 3rd party app that provides many storage backends for many common file storage APIs (including S3).

4.14.5 Upgrading from `django-staticfiles`

`django.contrib.staticfiles` began its life as django-staticfiles. If you're upgrading from django-staticfiles older than 1.0 (e.g. 0.3.4) to `django.contrib.staticfiles`, you'll need to make a few changes:

- Application files should now live in a `static` directory in each app (django-staticfiles used the name `media`, which was slightly confusing).
- The management commands `build_static` and `resolve_static` are now called `collectstatic` and `findstatic`.
- The settings `STATICFILES_PREPEND_LABEL_APPS`, `STATICFILES_MEDIA_DIRNAMES` and `STATICFILES_EXCLUDED_APPS` were removed.
- The setting `STATICFILES_RESOLVERS` was removed, and replaced by the new `STATICFILES_FINDERS`.
- The default for `STATICFILES_STORAGE` was renamed from `staticfiles.storage.StaticFileStorage` to `staticfiles.storage.StaticFilesStorage`
- If using *runserver* for local development (and the `DEBUG` setting is `True`), you no longer need to add anything to your URLconf for serving static files in development.

4.14.6 Learn more

This document has covered the basics and some common usage patterns. For complete details on all the settings, commands, template tags, and other pieces include in `django.contrib.staticfiles`, see *the staticfiles reference*.

See Also:

The Django community aggregator, where we aggregate content from the global Django community. Many writers in the aggregator write this sort of how-to material.

CHAPTER

FIVE

DJANGO FAQ

5.1 FAQ: General

5.1.1 Why does this project exist?

Django grew from a very practical need: World Online, a newspaper Web operation, is responsible for building intensive Web applications on journalism deadlines. In the fast-paced newsroom, World Online often has only a matter of hours to take a complicated Web application from concept to public launch.

At the same time, the World Online Web developers have consistently been perfectionists when it comes to following best practices of Web development.

In fall 2003, the World Online developers (Adrian Holovaty and Simon Willison) ditched PHP and began using Python to develop its Web sites. As they built intensive, richly interactive sites such as Lawrence.com, they began to extract a generic Web development framework that let them build Web applications more and more quickly. They tweaked this framework constantly, adding improvements over two years.

In summer 2005, World Online decided to open-source the resulting software, Django. Django would not be possible without a whole host of open-source projects – Apache, Python, and PostgreSQL to name a few – and we're thrilled to be able to give something back to the open-source community.

5.1.2 What does "Django" mean, and how do you pronounce it?

Django is named after Django Reinhardt, a gypsy jazz guitarist from the 1930s to early 1950s. To this day, he's considered one of the best guitarists of all time.

Listen to his music. You'll like it.

Django is pronounced **JANG**-oh. Rhymes with FANG-oh. The "D" is silent.

We've also recorded an audio clip of the pronunciation.

5.1.3 Is Django stable?

Yes, it's quite stable. World Online has been using Django for many years. Sites built on Django have weathered traffic spikes of over one million hits an hour.

5.1.4 Does Django scale?

Yes. Compared to development time, hardware is cheap, and so Django is designed to take advantage of as much hardware as you can throw at it.

Django uses a "shared-nothing" architecture, which means you can add hardware at any level – database servers, caching servers or Web/application servers.

The framework cleanly separates components such as its database layer and application layer. And it ships with a simple-yet-powerful *cache framework*.

5.1.5 Who's behind this?

Django was originally developed at World Online, the Web department of a newspaper in Lawrence, Kansas, USA. Django's now run by an international team of volunteers; you can read all about them over at the *list of committers*

5.1.6 Which sites use Django?

DjangoSites.org features a constantly growing list of Django-powered sites.

5.1.7 Django appears to be a MVC framework, but you call the Controller the "view", and the View the "template". How come you don't use the standard names?

Well, the standard names are debatable.

In our interpretation of MVC, the "view" describes the data that gets presented to the user. It's not necessarily *how* the data *looks*, but *which* data is presented. The view describes *which data you see*, not *how you see it.* It's a subtle distinction.

So, in our case, a "view" is the Python callback function for a particular URL, because that callback function describes which data is presented.

Furthermore, it's sensible to separate content from presentation – which is where templates come in. In Django, a "view" describes which data is presented, but a view normally delegates to a template, which describes *how* the data is presented.

Where does the "controller" fit in, then? In Django's case, it's probably the framework itself: the machinery that sends a request to the appropriate view, according to the Django URL configuration.

If you're hungry for acronyms, you might say that Django is a "MTV" framework – that is, "model", "template", and "view." That breakdown makes much more sense.

At the end of the day, of course, it comes down to getting stuff done. And, regardless of how things are named, Django gets stuff done in a way that's most logical to us.

5.1.8 <Framework X> does <feature Y> – why doesn't Django?

We're well aware that there are other awesome Web frameworks out there, and we're not averse to borrowing ideas where appropriate. However, Django was developed precisely because we were unhappy with the status quo, so please be aware that "because <Framework X> does it" is not going to be sufficient reason to add a given feature to Django.

5.1.9 Why did you write all of Django from scratch, instead of using other Python libraries?

When Django was originally written a couple of years ago, Adrian and Simon spent quite a bit of time exploring the various Python Web frameworks available.

In our opinion, none of them were completely up to snuff.

We're picky. You might even call us perfectionists. (With deadlines.)

Over time, we stumbled across open-source libraries that did things we'd already implemented. It was reassuring to see other people solving similar problems in similar ways, but it was too late to integrate outside code: We'd already written, tested and implemented our own framework bits in several production settings – and our own code met our needs delightfully.

In most cases, however, we found that existing frameworks/tools inevitably had some sort of fundamental, fatal flaw that made us squeamish. No tool fit our philosophies 100%.

Like we said: We're picky.

We've documented our philosophies on the *design philosophies page*.

5.1.10 Is Django a content-management-system (CMS)?

No, Django is not a CMS, or any sort of "turnkey product" in and of itself. It's a Web framework; it's a programming tool that lets you build Web sites.

For example, it doesn't make much sense to compare Django to something like Drupal, because Django is something you use to *create* things like Drupal.

Of course, Django's automatic admin site is fantastic and timesaving – but the admin site is one module of Django the framework. Furthermore, although Django has special conveniences for building "CMS-y" apps, that doesn't mean it's not just as appropriate for building "non-CMS-y" apps (whatever that means!).

5.1.11 How can I download the Django documentation to read it offline?

The Django docs are available in the `docs` directory of each Django tarball release. These docs are in reST (reStructuredText) format, and each text file corresponds to a Web page on the official Django site.

Because the documentation is stored in revision control, you can browse documentation changes just like you can browse code changes.

Technically, the docs on Django's site are generated from the latest development versions of those reST documents, so the docs on the Django site may offer more information than the docs that come with the latest Django release.

5.1.12 Where can I find Django developers for hire?

Consult our developers for hire page for a list of Django developers who would be happy to help you.

You might also be interested in posting a job to http://djangogigs.com/ . If you want to find Django-capable people in your local area, try http://djangopeople.net/ .

5.2 FAQ: Installation

5.2.1 How do I get started?

1. Download the code.
2. Install Django (read the *installation guide*).
3. Walk through the *tutorial*.
4. Check out the rest of the *documentation*, and ask questions if you run into trouble.

5.2.2 What are Django's prerequisites?

Django requires Python, specifically Python 2.6.5 - 2.7.x. No other Python libraries are required for basic Django usage.

For a development environment – if you just want to experiment with Django – you don't need to have a separate Web server installed; Django comes with its own lightweight development server. For a production environment, Django follows the WSGI spec, **PEP 3333**, which means it can run on a variety of server platforms. See *Deploying Django* for some popular alternatives. Also, the server arrangements wiki page contains details for several deployment strategies.

If you want to use Django with a database, which is probably the case, you'll also need a database engine. PostgreSQL is recommended, because we're PostgreSQL fans, and MySQL, SQLite 3, and Oracle are also supported.

5.2.3 Do I lose anything by using Python 2.6 versus newer Python versions, such as Python 2.7?

Not in the core framework. Currently, Django itself officially supports Python 2.6 (2.6.5 or higher) and 2.7. However, newer versions of Python are often faster, have more features, and are better supported. If you use a newer version of Python you will also have access to some APIs that aren't available under older versions of Python.

Third-party applications for use with Django are, of course, free to set their own version requirements.

Over the next year or two Django will begin dropping support for older Python versions as part of a migration which will end with Django running on Python 3 (see below for details).

All else being equal, we recommend that you use the latest 2.x release (currently Python 2.7). This will let you take advantage of the numerous improvements and optimizations to the Python language since version 2.6, and will help ease the process of dropping support for older Python versions on the road to Python 3.

5.2.4 What Python version can I use with Django?

Django version	Python versions
1.0	2.3, 2.4, 2.5, 2.6
1.1	2.3, 2.4, 2.5, 2.6
1.2	2.4, 2.5, 2.6, 2.7
1.3	2.4, 2.5, 2.6, 2.7
1.4	**2.5, 2.6, 2.7**
1.5 (future)	*2.6, 2.7, 3.x (experimental)*

5.2.5 Can I use Django with Python 3?

Not at the moment. Python 3.0 introduced a number of backwards-incompatible changes to the Python language, and although these changes are generally a good thing for Python's future, it will be a while before most Python software catches up and is able to run on Python 3.0. For larger Python-based software like Django, the transition is expected to take at least a year or two (since it involves dropping support for older Python releases and so must be done gradually).

In the meantime, Python 2.x releases will be supported and provided with bug fixes and security updates by the Python development team, so continuing to use a Python 2.x release during the transition should not present any risk.

5.2.6 Will Django run under shared hosting (like TextDrive or Dreamhost)?

See our Django-friendly Web hosts page.

5.2.7 Should I use the stable version or development version?

Generally, if you're using code in production, you should be using a stable release. The Django project publishes a full stable release every nine months or so, with bugfix updates in between. These stable releases contain the API that is covered by our backwards compatibility guarantees; if you write code against stable releases, you shouldn't have any problems upgrading when the next official version is released.

5.3 FAQ: Using Django

5.3.1 Why do I get an error about importing DJANGO_SETTINGS_MODULE?

Make sure that:

- The environment variable DJANGO_SETTINGS_MODULE is set to a fully-qualified Python module (i.e. "mysite.settings").
- Said module is on `sys.path` (`import mysite.settings` should work).
- The module doesn't contain syntax errors (of course).

5.3.2 I can't stand your template language. Do I have to use it?

We happen to think our template engine is the best thing since chunky bacon, but we recognize that choosing a template language runs close to religion. There's nothing about Django that requires using the template language, so if you're attached to ZPT, Cheetah, or whatever, feel free to use those.

5.3.3 Do I have to use your model/database layer?

Nope. Just like the template system, the model/database layer is decoupled from the rest of the framework.

The one exception is: If you use a different database library, you won't get to use Django's automatically-generated admin site. That app is coupled to the Django database layer.

5.3.4 How do I use image and file fields?

Using a `FileField` or an `ImageField` in a model takes a few steps:

1. In your settings file, you'll need to define `MEDIA_ROOT` as the full path to a directory where you'd like Django to store uploaded files. (For performance, these files are not stored in the database.) Define `MEDIA_URL` as the base public URL of that directory. Make sure that this directory is writable by the Web server's user account.
2. Add the `FileField` or `ImageField` to your model, making sure to define the `upload_to` option to tell Django to which subdirectory of `MEDIA_ROOT` it should upload files.
3. All that will be stored in your database is a path to the file (relative to `MEDIA_ROOT`). You'll most likely want to use the convenience `url` attribute provided by Django. For example, if your `ImageField` is called `mug_shot`, you can get the absolute path to your image in a template with `{{ object.mug_shot.url }}`.

5.3.5 How do I make a variable available to all my templates?

Sometimes your templates just all need the same thing. A common example would be dynamically-generated menus. At first glance, it seems logical to simply add a common dictionary to the template context.

The correct solution is to use a `RequestContext`. Details on how to do this are here: *Subclassing Context: RequestContext*.

5.4 FAQ: Getting Help

5.4.1 How do I do X? Why doesn't Y work? Where can I go to get help?

If this FAQ doesn't contain an answer to your question, you might want to try the django-users mailing list. Feel free to ask any question related to installing, using, or debugging Django.

If you prefer IRC, the #django IRC channel on the Freenode IRC network is an active community of helpful individuals who may be able to solve your problem.

5.4.2 Why hasn't my message appeared on django-users?

django-users has a lot of subscribers. This is good for the community, as it means many people are available to contribute answers to questions. Unfortunately, it also means that django-users is an attractive target for spammers.

In order to combat the spam problem, when you join the django-users mailing list, we manually moderate the first message you send to the list. This means that spammers get caught, but it also means that your first question to the list might take a little longer to get answered. We apologize for any inconvenience that this policy may cause.

5.4.3 Nobody on django-users answered my question! What should I do?

Try making your question more specific, or provide a better example of your problem.

As with most open-source mailing lists, the folks on django-users are volunteers. If nobody has answered your question, it may be because nobody knows the answer, it may be because nobody can understand the question, or it may be that everybody that can help is busy. One thing you might try is to ask the question on IRC – visit the #django IRC channel on the Freenode IRC network.

You might notice we have a second mailing list, called django-developers – but please don't email support questions to this mailing list. This list is for discussion of the development of Django itself. Asking a tech support question there is considered quite impolite.

5.4.4 I think I've found a bug! What should I do?

Detailed instructions on how to handle a potential bug can be found in our *Guide to contributing to Django*.

5.4.5 I think I've found a security problem! What should I do?

If you think you've found a security problem with Django, please send a message to security@djangoproject.com. This is a private list only open to long-time, highly trusted Django developers, and its archives are not publicly readable.

Due to the sensitive nature of security issues, we ask that if you think you have found a security problem, *please* don't send a message to one of the public mailing lists. Django has a *policy for handling security issues*; while a defect is outstanding, we would like to minimize any damage that could be inflicted through public knowledge of that defect.

5.5 FAQ: Databases and models

5.5.1 How can I see the raw SQL queries Django is running?

Make sure your Django `DEBUG` setting is set to `True`. Then, just do this:

```
>>> from django.db import connection
>>> connection.queries
[{'sql': 'SELECT polls_polls.id,polls_polls.question,polls_polls.pub_date FROM polls_polls',
'time': '0.002'}]
```

`connection.queries` is only available if `DEBUG` is `True`. It's a list of dictionaries in order of query execution. Each dictionary has the following:

```
``sql`` -- The raw SQL statement
``time`` -- How long the statement took to execute, in seconds.
```

`connection.queries` includes all SQL statements – INSERTs, UPDATES, SELECTs, etc. Each time your app hits the database, the query will be recorded. Note that the SQL recorded here may be *incorrectly quoted under SQLite*.

If you are using *multiple databases*, you can use the same interface on each member of the `connections` dictionary:

```
>>> from django.db import connections
>>> connections['my_db_alias'].queries
```

5.5.2 Can I use Django with a pre-existing database?

Yes. See *Integrating with a legacy database*.

5.5.3 If I make changes to a model, how do I update the database?

If you don't mind clearing data, your project's `manage.py` utility has a `flush` option to reset the database to the state it was in immediately after `syncdb` was executed.

If you do care about deleting data, you'll have to execute the `ALTER TABLE` statements manually in your database.

There are external projects which handle schema updates, of which the current defacto standard is south.

5.5.4 Do Django models support multiple-column primary keys?

No. Only single-column primary keys are supported.

But this isn't an issue in practice, because there's nothing stopping you from adding other constraints (using the `unique_together` model option or creating the constraint directly in your database), and enforcing the uniqueness at that level. Single-column primary keys are needed for things such as the admin interface to work; e.g., you need a simple way of being able to specify an object to edit or delete.

5.5.5 How do I add database-specific options to my CREATE TABLE statements, such as specifying MyISAM as the table type?

We try to avoid adding special cases in the Django code to accommodate all the database-specific options such as table type, etc. If you'd like to use any of these options, create an *SQL initial data file* that contains `ALTER TABLE` statements that do what you want to do. The initial data files are executed in your database after the `CREATE TABLE` statements.

For example, if you're using MySQL and want your tables to use the MyISAM table type, create an initial data file and put something like this in it:

```
ALTER TABLE myapp_mytable ENGINE=MyISAM;
```

As explained in the *SQL initial data file* documentation, this SQL file can contain arbitrary SQL, so you can make any sorts of changes you need to make.

5.5.6 Why is Django leaking memory?

Django isn't known to leak memory. If you find your Django processes are allocating more and more memory, with no sign of releasing it, check to make sure your `DEBUG` setting is set to `False`. If `DEBUG` is `True`, then Django saves a copy of every SQL statement it has executed.

(The queries are saved in `django.db.connection.queries`. See How can I see the raw SQL queries Django is running?.)

To fix the problem, set `DEBUG` to `False`.

If you need to clear the query list manually at any point in your functions, just call `reset_queries()`, like this:

```
from django import db
db.reset_queries()
```

5.6 FAQ: The admin

5.6.1 I can't log in. When I enter a valid username and password, it just brings up the login page again, with no error messages.

The login cookie isn't being set correctly, because the domain of the cookie sent out by Django doesn't match the domain in your browser. Try these two things:

- Set the SESSION_COOKIE_DOMAIN setting in your admin config file to match your domain. For example, if you're going to "http://www.example.com/admin/" in your browser, in "myproject.settings" you should set SESSION_COOKIE_DOMAIN = 'www.example.com'.
- Some browsers (Firefox?) don't like to accept cookies from domains that don't have dots in them. If you're running the admin site on "localhost" or another domain that doesn't have a dot in it, try going to "localhost.localdomain" or "127.0.0.1". And set SESSION_COOKIE_DOMAIN accordingly.

5.6.2 I can't log in. When I enter a valid username and password, it brings up the login page again, with a "Please enter a correct username and password" error.

If you're sure your username and password are correct, make sure your user account has is_active and is_staff set to True. The admin site only allows access to users with those two fields both set to True.

5.6.3 How can I prevent the cache middleware from caching the admin site?

Set the CACHE_MIDDLEWARE_ANONYMOUS_ONLY setting to True. See the *cache documentation* for more information.

5.6.4 How do I automatically set a field's value to the user who last edited the object in the admin?

The ModelAdmin class provides customization hooks that allow you to transform an object as it saved, using details from the request. By extracting the current user from the request, and customizing the save_model() hook, you can update an object to reflect the user that edited it. See *the documentation on ModelAdmin methods* for an example.

5.6.5 How do I limit admin access so that objects can only be edited by the users who created them?

The ModelAdmin class also provides customization hooks that allow you to control the visibility and editability of objects in the admin. Using the same trick of extracting the user from the request, the queryset() and has_change_permission() can be used to control the visibility and editability of objects in the admin.

5.6.6 My admin-site CSS and images showed up fine using the development server, but they're not displaying when using mod_wsgi.

See *serving the admin files* in the "How to use Django with mod_wsgi" documentation.

5.6.7 My "list_filter" contains a ManyToManyField, but the filter doesn't display.

Django won't bother displaying the filter for a ManyToManyField if there are fewer than two related objects.

For example, if your list_filter includes sites, and there's only one site in your database, it won't display a "Site" filter. In that case, filtering by site would be meaningless.

5.6.8 How can I customize the functionality of the admin interface?

You've got several options. If you want to piggyback on top of an add/change form that Django automatically generates, you can attach arbitrary JavaScript modules to the page via the model's `class Admin js` parameter. That parameter is a list of URLs, as strings, pointing to JavaScript modules that will be included within the admin form via a `<script>` tag.

If you want more flexibility than simply tweaking the auto-generated forms, feel free to write custom views for the admin. The admin is powered by Django itself, and you can write custom views that hook into the authentication system, check permissions and do whatever else they need to do.

If you want to customize the look-and-feel of the admin interface, read the next question.

5.6.9 The dynamically-generated admin site is ugly! How can I change it?

We like it, but if you don't agree, you can modify the admin site's presentation by editing the CSS stylesheet and/or associated image files. The site is built using semantic HTML and plenty of CSS hooks, so any changes you'd like to make should be possible by editing the stylesheet. We've got a *guide to the CSS used in the admin* to get you started.

5.6.10 What browsers are supported for using the admin?

The admin provides a fully-functional experience to YUI's A-grade browsers, with the notable exception of IE6, which is not supported.

There *may* be minor stylistic differences between supported browsers—for example, some browsers may not support rounded corners. These are considered acceptable variations in rendering.

5.7 FAQ: Contributing code

5.7.1 How can I get started contributing code to Django?

Thanks for asking! We've written an entire document devoted to this question. It's titled *Contributing to Django*.

5.7.2 I submitted a bug fix in the ticket system several weeks ago. Why are you ignoring my patch?

Don't worry: We're not ignoring you!

It's important to understand there is a difference between "a ticket is being ignored" and "a ticket has not been attended to yet." Django's ticket system contains hundreds of open tickets, of various degrees of impact on end-user functionality, and Django's developers have to review and prioritize.

On top of that: the people who work on Django are all volunteers. As a result, the amount of time that we have to work on the framework is limited and will vary from week to week depending on our spare time. If we're busy, we may not be able to spend as much time on Django as we might want.

The best way to make sure tickets do not get hung up on the way to checkin is to make it dead easy, even for someone who may not be intimately familiar with that area of the code, to understand the problem and verify the fix:

- Are there clear instructions on how to reproduce the bug? If this touches a dependency (such as PIL), a contrib module, or a specific database, are those instructions clear enough even for someone not familiar with it?

- If there are several patches attached to the ticket, is it clear what each one does, which ones can be ignored and which matter?
- Does the patch include a unit test? If not, is there a very clear explanation why not? A test expresses succinctly what the problem is, and shows that the patch actually fixes it.

If your patch stands no chance of inclusion in Django, we won't ignore it – we'll just close the ticket. So if your ticket is still open, it doesn't mean we're ignoring you; it just means we haven't had time to look at it yet.

5.7.3 When and how might I remind the core team of a patch I care about?

A polite, well-timed message to the mailing list is one way to get attention. To determine the right time, you need to keep an eye on the schedule. If you post your message when the core developers are trying to hit a feature deadline or manage a planning phase, you're not going to get the sort of attention you require. However, if you draw attention to a ticket when the core developers are paying particular attention to bugs – just before a bug fixing sprint, or in the lead up to a beta release for example – you're much more likely to get a productive response.

Gentle IRC reminders can also work – again, strategically timed if possible. During a bug sprint would be a very good time, for example.

Another way to get traction is to pull several related tickets together. When the core developers sit down to fix a bug in an area they haven't touched for a while, it can take a few minutes to remember all the fine details of how that area of code works. If you collect several minor bug fixes together into a similarly themed group, you make an attractive target, as the cost of coming up to speed on an area of code can be spread over multiple tickets.

Please refrain from emailing core developers personally, or repeatedly raising the same issue over and over. This sort of behavior will not gain you any additional attention – certainly not the attention that you need in order to get your pet bug addressed.

5.7.4 But I've reminded you several times and you keep ignoring my patch!

Seriously - we're not ignoring you. If your patch stands no chance of inclusion in Django, we'll close the ticket. For all the other tickets, we need to prioritize our efforts, which means that some tickets will be addressed before others.

One of the criteria that is used to prioritize bug fixes is the number of people that will likely be affected by a given bug. Bugs that have the potential to affect many people will generally get priority over those that are edge cases.

Another reason that bugs might be ignored for while is if the bug is a symptom of a larger problem. While we can spend time writing, testing and applying lots of little patches, sometimes the right solution is to rebuild. If a rebuild or refactor of a particular component has been proposed or is underway, you may find that bugs affecting that component will not get as much attention. Again, this is just a matter of prioritizing scarce resources. By concentrating on the rebuild, we can close all the little bugs at once, and hopefully prevent other little bugs from appearing in the future.

Whatever the reason, please keep in mind that while you may hit a particular bug regularly, it doesn't necessarily follow that every single Django user will hit the same bug. Different users use Django in different ways, stressing different parts of the code under different conditions. When we evaluate the relative priorities, we are generally trying to consider the needs of the entire community, not just the severity for one particular user. This doesn't mean that we think your problem is unimportant – just that in the limited time we have available, we will always err on the side of making 10 people happy rather than making 1 person happy.

CHAPTER

SIX

API REFERENCE

6.1 Authentication backends

This document details the authentication backends that come with Django. For information on how to use them and how to write your own authentication backends, see the *Other authentication sources section* of the *User authentication guide*.

6.1.1 Available authentication backends

The following backends are available in `django.contrib.auth.backends`:

class **`ModelBackend`**
This is the default authentication backend used by Django. It authenticates using usernames and passwords stored in the `User` model.

class **`RemoteUserBackend`**
Use this backend to take advantage of external-to-Django-handled authentication. It authenticates using usernames passed in `request.META['REMOTE_USER']`. See the *Authenticating against REMOTE_USER* documentation.

6.2 Class-based views

Class-based views API reference. For introductory material, see *Class-based views*.

6.2.1 Base views

The following three classes provide much of the functionality needed to create Django views. You may think of them as *parent* views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins and Generic class-based views.

class `django.views.generic.base.`**`View`**
The master class-based base view. All other class-based views inherit from this base class.

Method Flowchart

1.`dispatch()`

2.`http_method_not_allowed()`

Example views.py:

```
from django.http import HttpResponse
from django.views.generic import View

class MyView(View):

    def get(self, request, *args, **kwargs):
        return HttpResponse('Hello, World!')
```

Example urls.py:

```
from django.conf.urls import patterns, url

from myapp.views import MyView

urlpatterns = patterns('',
    url(r'^mine/$', MyView.as_view(), name='my-view'),
)
```

Methods

dispatch(*request*, **args*, ***kwargs*)

The `view` part of the view – the method that accepts a `request` argument plus arguments, and returns a HTTP response.

The default implementation will inspect the HTTP method and attempt to delegate to a method that matches the HTTP method; a `GET` will be delegated to `get()`, a `POST` to `post()`, and so on.

The default implementation also sets `request`, `args` and `kwargs` as instance variables, so any method on the view can know the full details of the request that was made to invoke the view.

http_method_not_allowed(*request*, **args*, ***kwargs*)

If the view was called with a HTTP method it doesn't support, this method is called instead.

The default implementation returns `HttpResponseNotAllowed` with list of allowed methods in plain text.

Note: Documentation on class-based views is a work in progress. As yet, only the methods defined directly on the class are documented here, not methods defined on superclasses.

---

class `django.views.generic.base.`**TemplateView**

Renders a given template, passing it a `{{ params }}` template variable, which is a dictionary of the parameters captured in the URL.

Ancestors (MRO)

- `django.views.generic.base.TemplateView`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.base.View`

Method Flowchart

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_context_data()`

Example views.py:

```
from django.views.generic.base import TemplateView

from articles.models import Article

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super(HomePageView, self).get_context_data(**kwargs)
        context['latest_articles'] = Article.objects.all()[:5]
        return context
```

Example urls.py:

```
from django.conf.urls import patterns, url

from myapp.views import HomePageView

urlpatterns = patterns('',
    url(r'^$', HomePageView.as_view(), name='home'),
)
```

Methods and Attributes

template_name
The full name of a template to use.

get_context_data(***kwargs*)
Return a context data dictionary consisting of the contents of `kwargs` stored in the context variable `params`.

Context

- `params`: The dictionary of keyword arguments captured from the URL pattern that served the view.

Note: Documentation on class-based views is a work in progress. As yet, only the methods defined directly on the class are documented here, not methods defined on superclasses.

---

class `django.views.generic.base.`**RedirectView**
Redirects to a given URL.

The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL. Because keyword interpolation is *always* done (even if no arguments are passed in), any `"%"` characters in the URL must be written as `"%%"` so that Python will convert them to a single percent sign on output.

If the given URL is `None`, Django will return an `HttpResponseGone` (410).

Ancestors (MRO)

- `django.views.generic.base.View`

Method Flowchart

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_redirect_url()`

Example views.py:

```
from django.shortcuts import get_object_or_404
from django.views.generic.base import RedirectView

from articles.models import Article

class ArticleCounterRedirectView(RedirectView):

    permanent = False
    query_string = True

    def get_redirect_url(self, pk):
        article = get_object_or_404(Article, pk=pk)
        article.update_counter()
        return reverse('product_detail', args=(pk,))
```

Example urls.py:

```
from django.conf.urls import patterns, url
from django.views.generic.base import RedirectView

from article.views import ArticleCounterRedirectView

urlpatterns = patterns('',

    url(r'r^(?P<pk>\d+)/$', ArticleCounterRedirectView.as_view(), name='article-counter'),
    url(r'^go-to-django/$', RedirectView.as_view(url='http://djangoproject.com'), name='go-to-dj
)
```

Methods and Attributes

url
The URL to redirect to, as a string. Or `None` to raise a 410 (Gone) HTTP error.

permanent
Whether the redirect should be permanent. The only difference here is the HTTP status code returned. If `True`, then the redirect will use status code 301. If `False`, then the redirect will use status code 302. By default, `permanent` is `True`.

query_string
Whether to pass along the GET query string to the new location. If `True`, then the query string is appended to the URL. If `False`, then the query string is discarded. By default, `query_string` is `False`.

get_redirect_url(***kwargs*)
Constructs the target URL for redirection.

The default implementation uses `url` as a starting string, performs expansion of `%` parameters in that string, as well as the appending of query string if requested by `query_string`. Subclasses may implement any behavior they wish, as long as the method returns a redirect-ready URL string.

Note: Documentation on class-based views is a work in progress. As yet, only the methods defined directly on the class are documented here, not methods defined on superclasses.

6.2.2 Generic display views

The two following generic class-based views are designed to display data. On many projects they are typically the most commonly used views.

class django.views.generic.detail.**DetailView**

While this view is executing, `self.object` will contain the object that the view is operating upon.

Ancestors (MRO)

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View

Method Flowchart

1. dispatch()
2. http_method_not_allowed()
3. get_template_names()
4. get_slug_field()
5. get_queryset()
6. get_object()
7. get_context_object_name()
8. get_context_data()
9. get()
10. render_to_response()

Example views.py:

```
from django.views.generic.detail import DetailView
from django.utils import timezone

from articles.models import Article

class ArticleDetailView(DetailView):

    model = Article

    def get_context_data(self, **kwargs):
        context = super(ArticleDetailView, self).get_context_data(**kwargs)
        context['now'] = timezone.now()
        return context
```

Example urls.py:

```
from django.conf.urls import patterns, url

from article.views import ArticleDetailView

urlpatterns = patterns('',
    url(r'^(?P<slug>[-_\w]+)/$', ArticleDetailView.as_view(), name='article-detail'),
)
```

class django.views.generic.list.**ListView**

A page representing a list of objects.

While this view is executing, `self.object_list` will contain the list of objects (usually, but not necessarily a queryset) that the view is operating upon.

Mixins

- `django.views.generic.list.ListView`
- `django.views.generic.list.MultipleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.list.BaseListView`
- `django.views.generic.list.MultipleObjectMixin`
- `django.views.generic.base.View`

Method Flowchart

1. `dispatch()`
2. `http_method_not_allowed()`
3. `get_template_names()`
4. `get_queryset()`
5. `get_objects()`
6. `get_context_data()`
7. `get()`
8. `render_to_response()`

6.2.3 Generic editing views

The views described here provide a foundation for editing content.

class `django.views.generic.edit.`**`FormView`**

A view that displays a form. On error, redisplays the form with validation errors; on success, redirects to a new URL.

Ancestors (MRO)

- `django.views.generic.edit.FormView`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.edit.BaseFormView`
- `django.views.generic.edit.FormMixin`
- `django.views.generic.edit.ProcessFormView`
- `django.views.generic.base.View`

class `django.views.generic.edit.`**`CreateView`**

A view that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.

Ancestors (MRO)

- `django.views.generic.edit.CreateView`
- `django.views.generic.detail.SingleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`

- django.views.generic.edit.BaseCreateView
- django.views.generic.edit.ModelFormMixin
- django.views.generic.edit.FormMixin
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.edit.ProcessFormView
- django.views.generic.base.View

class django.views.generic.edit.**UpdateView**

A view that displays a form for editing an existing object, redisplaying the form with validation errors (if there are any) and saving changes to the object. This uses a form automatically generated from the object's model class (unless a form class is manually specified).

Ancestors (MRO)

- django.views.generic.edit.UpdateView
- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseUpdateView
- django.views.generic.edit.ModelFormMixin
- django.views.generic.edit.FormMixin
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.edit.ProcessFormView
- django.views.generic.base.View

class django.views.generic.edit.**DeleteView**

A view that displays a confirmation page and deletes an existing object. The given object will only be deleted if the request method is POST. If this view is fetched via GET, it will display a confirmation page that should contain a form that POSTs to the same URL.

Ancestors (MRO)

- django.views.generic.edit.DeleteView
- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.edit.BaseDeleteView
- django.views.generic.edit.DeletionMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View

Notes

- The delete confirmation page displayed to a GET request uses a template_name_suffix of '_confirm_delete'.

6.2.4 Generic date views

Date-based generic views (in the module `django.views.generic.dates`) are views for displaying drilldown pages for date-based data.

class `django.views.generic.dates.`**`ArchiveIndexView`**

A top-level index page showing the "latest" objects, by date. Objects with a date in the *future* are not included unless you set `allow_future` to `True`.

Ancestors (MRO)

- `django.views.generic.dates.ArchiveIndexView`
- `django.views.generic.list.MultipleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.dates.BaseArchiveIndexView`
- `django.views.generic.dates.BaseDateListView`
- `django.views.generic.list.MultipleObjectMixin`
- `django.views.generic.dates.DateMixin`
- `django.views.generic.base.View`

Notes

- Uses a default `context_object_name` of `latest`.
- Uses a default `template_name_suffix` of `_archive`.

class `django.views.generic.dates.`**`YearArchiveView`**

A yearly archive page showing all available months in a given year. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Ancestors (MRO)

- `django.views.generic.dates.YearArchiveView`
- `django.views.generic.list.MultipleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.dates.BaseYearArchiveView`
- `django.views.generic.dates.YearMixin`
- `django.views.generic.dates.BaseDateListView`
- `django.views.generic.list.MultipleObjectMixin`
- `django.views.generic.dates.DateMixin`
- `django.views.generic.base.View`

`make_object_list`

A boolean specifying whether to retrieve the full list of objects for this year and pass those to the template. If `True`, the list of objects will be made available to the context. By default, this is `False`.

`get_make_object_list``()`

Determine if an object list will be returned as part of the context. If `False`, the `None` queryset will be used as the object list.

Context

In addition to the context provided by `django.views.generic.list.MultipleObjectMixin` (via `django.views.generic.dates.BaseDateListView`), the template's context will be:

- `date_list`: A `DateQuerySet` object object containing all months that have objects available according to `queryset`, represented as `datetime.datetime` objects, in ascending order.
- `year`: A `datetime.date` object representing the given year.
- `next_year`: A `datetime.date` object representing the first day of the next year. If the next year is in the future, this will be `None`.
- `previous_year`: A `datetime.date` object representing the first day of the previous year. Unlike `next_year`, this will never be `None`.

Notes

- Uses a default `template_name_suffix` of `_archive_year`.

class `django.views.generic.dates.`**`MonthArchiveView`**

A monthly archive page showing all objects in a given month. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Ancestors (MRO)

- `django.views.generic.dates.MonthArchiveView`
- `django.views.generic.list.MultipleObjectTemplateResponseMixin`
- `django.views.generic.base.TemplateResponseMixin`
- `django.views.generic.dates.BaseMonthArchiveView`
- `django.views.generic.dates.YearMixin`
- `django.views.generic.dates.MonthMixin`
- `django.views.generic.dates.BaseDateListView`
- `django.views.generic.list.MultipleObjectMixin`
- `django.views.generic.dates.DateMixin`
- `django.views.generic.base.View`

Context

In addition to the context provided by `MultipleObjectMixin` (via `BaseDateListView`), the template's context will be:

- `date_list`: A `DateQuerySet` object containing all days that have objects available in the given month, according to `queryset`, represented as `datetime.datetime` objects, in ascending order.
- `month`: A `datetime.date` object representing the given month.
- `next_month`: A `datetime.date` object representing the first day of the next month. If the next month is in the future, this will be `None`.
- `previous_month`: A `datetime.date` object representing the first day of the previous month. Unlike `next_month`, this will never be `None`.

Notes

- Uses a default `template_name_suffix` of `_archive_month`.

class django.views.generic.dates.**WeekArchiveView**

A weekly archive page showing all objects in a given week. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Ancestors (MRO)

- django.views.generic.dates.WeekArchiveView
- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseWeekArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.WeekMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

Context

In addition to the context provided by `MultipleObjectMixin` (via `BaseDateListView`), the template's context will be:

- `week`: A `datetime.date` object representing the first day of the given week.
- `next_week`: A `datetime.date` object representing the first day of the next week. If the next week is in the future, this will be `None`.
- `previous_week`: A `datetime.date` object representing the first day of the previous week. Unlike `next_week`, this will never be `None`.

Notes

- Uses a default `template_name_suffix` of `_archive_week`.

class django.views.generic.dates.**DayArchiveView**

A day archive page showing all objects in a given day. Days in the future throw a 404 error, regardless of whether any objects exist for future days, unless you set `allow_future` to `True`.

Ancestors (MRO)

- django.views.generic.dates.DayArchiveView
- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseDayArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.MonthMixin
- django.views.generic.dates.DayMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin

- django.views.generic.base.View

Context

In addition to the context provided by MultipleObjectMixin (via BaseDateListView), the template's context will be:

- day: A datetime.date object representing the given day.
- next_day: A datetime.date object representing the next day. If the next day is in the future, this will be None.
- previous_day: A datetime.date object representing the previous day. Unlike next_day, this will never be None.
- next_month: A datetime.date object representing the first day of the next month. If the next month is in the future, this will be None.
- previous_month: A datetime.date object representing the first day of the previous month. Unlike next_month, this will never be None.

Notes

- Uses a default template_name_suffix of _archive_day.

class django.views.generic.dates.**TodayArchiveView**

A day archive page showing all objects for *today*. This is exactly the same as django.views.generic.dates.DayArchiveView, except today's date is used instead of the year/month/day arguments.

Ancestors (MRO)

- django.views.generic.dates.TodayArchiveView
- django.views.generic.list.MultipleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.dates.BaseTodayArchiveView
- django.views.generic.dates.BaseDayArchiveView
- django.views.generic.dates.YearMixin
- django.views.generic.dates.MonthMixin
- django.views.generic.dates.DayMixin
- django.views.generic.dates.BaseDateListView
- django.views.generic.list.MultipleObjectMixin
- django.views.generic.dates.DateMixin
- django.views.generic.base.View

class django.views.generic.dates.**DateDetailView**

A page representing an individual object. If the object has a date value in the future, the view will throw a 404 error by default, unless you set allow_future to True.

Ancestors (MRO)

- django.views.generic.dates.DateDetailView
- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin

- `django.views.generic.dates.BaseDateDetailView`
- `django.views.generic.dates.YearMixin`
- `django.views.generic.dates.MonthMixin`
- `django.views.generic.dates.DayMixin`
- `django.views.generic.dates.DateMixin`
- `django.views.generic.detail.BaseDetailView`
- `django.views.generic.detail.SingleObjectMixin`
- `django.views.generic.base.View`

Note: All of the generic views listed above have matching Base* views that only differ in that the they do not include the `SingleObjectTemplateResponseMixin`.

6.2.5 Class-based views mixins

Class-based views API reference. For introductory material, see *Using mixins with class-based views*.

Simple mixins

class `django.views.generic.base.`**`ContextMixin`**

New in version 1.5: *Please see the release notes* **classpath**

`django.views.generic.base.ContextMixin`

Methods

`get_context_data`(***kwargs*)

Returns a dictionary representing the template context. The keyword arguments provided will make up the returned context.

class `django.views.generic.base.`**`TemplateResponseMixin`**

Provides a mechanism to construct a `TemplateResponse`, given suitable context. The template to use is configurable and can be further customized by subclasses.

Methods and Attributes

`response_class`

The response class to be returned by `render_to_response` method. Default is `TemplateResponse`. The template and context of `TemplateResponse` instances can be altered later (e.g. in *template response middleware*).

If you need custom template loading or custom context object instantiation, create a `TemplateResponse` subclass and assign it to `response_class`.

`render_to_response`(*context*, ***response_kwargs*)

Returns a `self.response_class` instance.

If any keyword arguments are provided, they will be passed to the constructor of the response class.

Calls `get_template_names()` to obtain the list of template names that will be searched looking for an existent template.

get_template_names()
Returns a list of template names to search for when rendering the template.

If `TemplateResponseMixin.template_name` is specified, the default implementation will return a list containing `TemplateResponseMixin.template_name` (if it is specified).

Single object mixins

class `django.views.generic.detail.`**SingleObjectMixin**
Provides a mechanism for looking up an object associated with the current HTTP request.

Methods and Attributes

model
The model that this view will display data for. Specifying `model = Foo` is effectively the same as specifying `queryset = Foo.objects.all()`.

queryset
A `QuerySet` that represents the objects. If provided, the value of `SingleObjectMixin.queryset` supersedes the value provided for `SingleObjectMixin.model`.

slug_field
The name of the field on the model that contains the slug. By default, `slug_field` is `'slug'`.

slug_url_kwarg
New in version 1.4: *Please see the release notes* The name of the URLConf keyword argument that contains the slug. By default, `slug_url_kwarg` is `'slug'`.

pk_url_kwarg
New in version 1.4: *Please see the release notes* The name of the URLConf keyword argument that contains the primary key. By default, `pk_url_kwarg` is `'pk'`.

context_object_name
Designates the name of the variable to use in the context.

get_object(*queryset=None*)
Returns the single object that this view will display. If `queryset` is provided, that queryset will be used as the source of objects; otherwise, `get_queryset()` will be used. `get_object()` looks for a `SingleObjectMixin.pk_url_kwarg` argument in the arguments to the view; if this argument is found, this method performs a primary-key based lookup using that value. If this argument is not found, it looks for a `SingleObjectMixin.slug_url_kwarg` argument, and performs a slug lookup using the `SingleObjectMixin.slug_field`.

get_queryset()
Returns the queryset that will be used to retrieve the object that this view will display. By default, `get_queryset()` returns the value of the `queryset` attribute if it is set, otherwise it constructs a `QuerySet` by calling the *all()* method on the `model` attribute's default manager.

get_context_object_name(*obj*)
Return the context variable name that will be used to contain the data that this view is manipulating. If `context_object_name` is not set, the context name will be constructed from the `object_name` of the model that the queryset is composed from. For example, the model `Article` would have context object named `'article'`.

get_context_data(***kwargs*)
Returns context data for displaying the list of objects.

Context

- `object`: The object that this view is displaying. If `context_object_name` is specified, that variable will also be set in the context, with the same value as `object`.

class `django.views.generic.detail.`**`SingleObjectTemplateResponseMixin`**

A mixin class that performs template-based response rendering for views that operate upon a single object instance. Requires that the view it is mixed with provides `self.object`, the object instance that the view is operating on. `self.object` will usually be, but is not required to be, an instance of a Django model. It may be `None` if the view is in the process of constructing a new instance.

Extends

- `TemplateResponseMixin`

Methods and Attributes

`template_name_field`

The field on the current object instance that can be used to determine the name of a candidate template. If either `template_name_field` itself or the value of the `template_name_field` on the current object instance is `None`, the object will not be used for a candidate template name.

`template_name_suffix`

The suffix to append to the auto-generated candidate template name. Default suffix is `_detail`.

`get_template_names``()`

Returns a list of candidate template names. Returns the following list:

- the value of `template_name` on the view (if provided)
- the contents of the `template_name_field` field on the object instance that the view is operating upon (if available)
- `<app_label>/<object_name><template_name_suffix>.html`

Multiple object mixins

class `django.views.generic.list.`**`MultipleObjectMixin`**

A mixin that can be used to display a list of objects.

If `paginate_by` is specified, Django will paginate the results returned by this. You can specify the page number in the URL in one of two ways:

- Use the `page` parameter in the URLconf. For example, this is what your URLconf might look like:

```
(r'^objects/page(?P<page>[0-9]+)/$', PaginatedView.as_view())
```

- Pass the page number via the `page` query-string parameter. For example, a URL would look like this:

```
/objects/?page=3
```

These values and lists are 1-based, not 0-based, so the first page would be represented as page `1`.

For more on pagination, read the *pagination documentation*.

As a special case, you are also permitted to use `last` as a value for `page`:

```
/objects/?page=last
```

This allows you to access the final page of results without first having to determine how many pages there are.

Note that `page` *must* be either a valid page number or the value `last`; any other value for `page` will result in a 404 error.

Extends

- `django.views.generic.base.ContextMixin`

Methods and Attributes

`allow_empty`
A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `True`.

`model`
The model that this view will display data for. Specifying `model = Foo` is effectively the same as specifying `queryset = Foo.objects.all()`.

`queryset`
A `QuerySet` that represents the objects. If provided, the value of `MultipleObjectMixin.queryset` supersedes the value provided for `MultipleObjectMixin.model`.

`paginate_by`
An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with `MultipleObjectMixin.paginate_by` objects per page. The view will expect either a `page` query string parameter (via `GET`) or a `page` variable specified in the URLconf.

`paginator_class`
The paginator class to be used for pagination. By default, `django.core.paginator.Paginator` is used. If the custom paginator class doesn't have the same constructor interface as `django.core.paginator.Paginator`, you will also need to provide an implementation for `MultipleObjectMixin.get_paginator()`.

`context_object_name`
Designates the name of the variable to use in the context.

`get_queryset``()`
Returns the queryset that represents the data this view will display.

`paginate_queryset`(*queryset*, *page_size*)
Returns a 4-tuple containing (`paginator`, `page`, `object_list`, `is_paginated`).

Constructed by paginating `queryset` into pages of size `page_size`. If the request contains a `page` argument, either as a captured URL argument or as a GET argument, `object_list` will correspond to the objects from that page.

`get_paginate_by`(*queryset*)
Returns the number of items to paginate by, or `None` for no pagination. By default this simply returns the value of `MultipleObjectMixin.paginate_by`.

`get_paginator`(*queryset*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)
Returns an instance of the paginator to use for this view. By default, instantiates an instance of `paginator_class`.

`get_allow_empty``()`
Return a boolean specifying whether to display the page if no objects are available. If this method returns `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `True`.

`get_context_object_name`(*object_list*)
Return the context variable name that will be used to contain the list of data that this view is manipulating. If `object_list` is a queryset of Django objects and `context_object_name` is not set, the context name will be the `object_name` of the model that the queryset is composed from, with postfix `'_list'` appended. For example, the model `Article` would have a context object named `article_list`.

get_context_data(***kwargs*)
Returns context data for displaying the list of objects.

Context

- `object_list`: The list of objects that this view is displaying. If `context_object_name` is specified, that variable will also be set in the context, with the same value as `object_list`.
- `is_paginated`: A boolean representing whether the results are paginated. Specifically, this is set to `False` if no page size has been specified, or if the available objects do not span multiple pages.
- `paginator`: An instance of `django.core.paginator.Paginator`. If the page is not paginated, this context variable will be `None`.
- `page_obj`: An instance of `django.core.paginator.Page`. If the page is not paginated, this context variable will be `None`.

class `django.views.generic.list.`**MultipleObjectTemplateResponseMixin**
A mixin class that performs template-based response rendering for views that operate upon a list of object instances. Requires that the view it is mixed with provides `self.object_list`, the list of object instances that the view is operating on. `self.object_list` may be, but is not required to be, a `QuerySet`.

Extends

- `TemplateResponseMixin`

Methods and Attributes

template_name_suffix
The suffix to append to the auto-generated candidate template name. Default suffix is `_list`.

get_template_names()
Returns a list of candidate template names. Returns the following list:

- the value of `template_name` on the view (if provided)
- `<app_label>/<object_name><template_name_suffix>.html`

Editing mixins

class `django.views.generic.edit.`**FormMixin**
A mixin class that provides facilities for creating and displaying forms.

Methods and Attributes

initial
A dictionary containing initial data for the form.

form_class
The form class to instantiate.

success_url
The URL to redirect to when the form is successfully processed.

get_initial()
Retrieve initial data for the form. By default, returns a copy of `initial`. Changed in version 1.4: In Django 1.3, this method was returning the `initial` class variable itself.

get_form_class()
Retrieve the form class to instantiate. By default `form_class`.

get_form(*form_class*)
Instantiate an instance of `form_class` using `get_form_kwargs()`.

get_form_kwargs()
Build the keyword arguments required to instantiate the form.

The `initial` argument is set to `get_initial()`. If the request is a `POST` or `PUT`, the request data (`request.POST` and `request.FILES`) will also be provided.

get_success_url()
Determine the URL to redirect to when the form is successfully validated. Returns `success_url` by default.

form_valid(*form*)
Redirects to `get_success_url()`.

form_invalid(*form*)
Renders a response, providing the invalid form as context.

get_context_data(***kwargs*)
Populates a context containing the contents of `kwargs`.

Context

- `form`: The form instance that was generated for the view.

Note: Views mixing `FormMixin` must provide an implementation of `form_valid()` and `form_invalid()`.

---

class `django.views.generic.edit.`**ModelFormMixin**
A form mixin that works on ModelForms, rather than a standalone form.

Since this is a subclass of `SingleObjectMixin`, instances of this mixin have access to the `model` and `queryset` attributes, describing the type of object that the ModelForm is manipulating. The view also provides `self.object`, the instance being manipulated. If the instance is being created, `self.object` will be `None`.

Mixins

- `django.views.generic.edit.FormMixin`
- `django.views.generic.detail.SingleObjectMixin`

Methods and Attributes

success_url
The URL to redirect to when the form is successfully processed.

`success_url` may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use `success_url="/polls/%(slug)s/"` to redirect to a URL composed out of the `slug` field on a model.

get_form_class()
Retrieve the form class to instantiate. If `FormMixin.form_class` is provided, that class will be used. Otherwise, a ModelForm will be instantiated using the model associated with the `queryset`, or with the `model`, depending on which attribute is provided.

get_form_kwargs()
Add the current instance (`self.object`) to the standard `FormMixin.get_form_kwargs()`.

get_success_url()
Determine the URL to redirect to when the form is successfully validated. Returns `ModelFormMixin.success_url` if it is provided; otherwise, attempts to use the `get_absolute_url()` of the object.

form_valid(*form*)
Saves the form instance, sets the current object for the view, and redirects to `get_success_url()`.

form_invalid()
Renders a response, providing the invalid form as context.

class `django.views.generic.edit.`**ProcessFormView**
A mixin that provides basic HTTP GET and POST workflow.

Note: This is named 'ProcessFormView' and inherits directly from `django.views.generic.base.View`, but breaks if used independently, so it is more of a mixin.

Extends

- `django.views.generic.base.View`

Methods and Attributes

get(*request*, **args*, ***kwargs*)
Constructs a form, then renders a response using a context that contains that form.

post(*request*, **args*, ***kwargs*)
Constructs a form, checks the form for validity, and handles it accordingly.

The PUT action is also handled, as an analog of POST.

class `django.views.generic.edit.`**DeletionMixin**
Enables handling of the `DELETE` http action.

Methods and Attributes

success_url
The url to redirect to when the nominated object has been successfully deleted.

get_success_url(*obj*)
Returns the url to redirect to when the nominated object has been successfully deleted. Returns `success_url` by default.

Date-based mixins

class `django.views.generic.dates.`**YearMixin**
A mixin that can be used to retrieve and provide parsing information for a year component of a date.

Methods and Attributes

year_format
The `strftime()` format to use when parsing the year. By default, this is `'%Y'`.

year
Optional The value for the year (as a string). By default, set to `None`, which means the year will be determined using other means.

get_year_format()
Returns the `strftime()` format to use when parsing the year. Returns `YearMixin.year_format` by default.

get_year()
Returns the year for which this view will display data. Tries the following sources, in order:

- The value of the `YearMixin.year` attribute.
- The value of the *year* argument captured in the URL pattern

- The value of the *year* GET query argument.

Raises a 404 if no valid year specification can be found.

class `django.views.generic.dates.`**MonthMixin**

A mixin that can be used to retrieve and provide parsing information for a month component of a date.

Methods and Attributes

month_format

The `strftime()` format to use when parsing the month. By default, this is `'%b'`.

month

Optional The value for the month (as a string). By default, set to `None`, which means the month will be determined using other means.

get_month_format()

Returns the `strftime()` format to use when parsing the month. Returns `MonthMixin.month_format` by default.

get_month()

Returns the month for which this view will display data. Tries the following sources, in order:

- The value of the `MonthMixin.month` attribute.
- The value of the *month* argument captured in the URL pattern
- The value of the *month* GET query argument.

Raises a 404 if no valid month specification can be found.

get_next_month(*date*)

Returns a date object containing the first day of the month after the date provided. Returns `None` if mixed with a view that sets `allow_future = False`, and the next month is in the future. If `allow_empty = False`, returns the next month that contains data.

get_prev_month(*date*)

Returns a date object containing the first day of the month before the date provided. If `allow_empty = False`, returns the previous month that contained data.

class `django.views.generic.dates.`**DayMixin**

A mixin that can be used to retrieve and provide parsing information for a day component of a date.

Methods and Attributes

day_format

The `strftime()` format to use when parsing the day. By default, this is `'%d'`.

day

Optional The value for the day (as a string). By default, set to `None`, which means the day will be determined using other means.

get_day_format()

Returns the `strftime()` format to use when parsing the day. Returns `DayMixin.day_format` by default.

get_day()

Returns the day for which this view will display data. Tries the following sources, in order:

- The value of the `DayMixin.day` attribute.
- The value of the *day* argument captured in the URL pattern
- The value of the *day* GET query argument.

Raises a 404 if no valid day specification can be found.

get_next_day(*date*)
Returns a date object containing the next day after the date provided. Returns `None` if mixed with a view that sets `allow_future = False`, and the next day is in the future. If `allow_empty = False`, returns the next day that contains data.

get_prev_day(*date*)
Returns a date object containing the previous day. If `allow_empty = False`, returns the previous day that contained data.

class `django.views.generic.dates.`**WeekMixin**
A mixin that can be used to retrieve and provide parsing information for a week component of a date.

Methods and Attributes

week_format
The `strftime()` format to use when parsing the week. By default, this is `'%U'`.

week
Optional The value for the week (as a string). By default, set to `None`, which means the week will be determined using other means.

get_week_format()
Returns the `strftime()` format to use when parsing the week. Returns `WeekMixin.week_format` by default.

get_week()
Returns the week for which this view will display data. Tries the following sources, in order:

- The value of the `WeekMixin.week` attribute.
- The value of the *week* argument captured in the URL pattern
- The value of the *week* GET query argument.

Raises a 404 if no valid week specification can be found.

class `django.views.generic.dates.`**DateMixin**
A mixin class providing common behavior for all date-based views.

Methods and Attributes

date_field
The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

When *time zone support* is enabled and `date_field` is a `DateTimeField`, dates are assumed to be in the current time zone. Otherwise, the queryset could include objects from the previous or the next day in the end user's time zone.

> **Warning:** In this situation, if you have implemented per-user time zone selection, the same URL may show a different set of objects, depending on the end user's time zone. To avoid this, you should use a `DateField` as the `date_field` attribute.

allow_future
A boolean specifying whether to include "future" objects on this page, where "future" means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

get_date_field()
Returns the name of the field that contains the date data that this view will operate on. Returns `DateMixin.date_field` by default.

get_allow_future()
Determine whether to include "future" objects on this page, where "future" means objects in which the field specified in `date_field` is greater than the current date/time. Returns `DateMixin.allow_future` by default.

class `django.views.generic.dates.`**BaseDateListView**
A base class that provides common behavior for all date-based views. There won't normally be a reason to instantiate `BaseDateListView`; instantiate one of the subclasses instead.

While this view (and it's subclasses) are executing, `self.object_list` will contain the list of objects that the view is operating upon, and `self.date_list` will contain the list of dates for which data is available.

Mixins

- `DateMixin`
- `MultipleObjectMixin`

Methods and Attributes

allow_empty
A boolean specifying whether to display the page if no objects are available. If this is `True` and no objects are available, the view will display an empty page instead of raising a 404. By default, this is `False`.

get_dated_items():
Returns a 3-tuple containing (`date_list`, `object_list`, `extra_context`).

`date_list` is the list of dates for which data is available. `object_list` is the list of objects. `extra_context` is a dictionary of context data that will be added to any context data provided by the `MultipleObjectMixin`.

get_dated_queryset(***lookup*)
Returns a queryset, filtered using the query arguments defined by `lookup`. Enforces any restrictions on the queryset, such as `allow_empty` and `allow_future`.

get_date_list(*queryset*, *date_type*)
Returns the list of dates of type `date_type` for which `queryset` contains entries. For example, `get_date_list(qs, 'year')` will return the list of years for which `qs` has entries. See `dates()` for the ways that the `date_type` argument can be used.

6.2.6 Specification

Each request served by a class-based view has an independent state; therefore, it is safe to store state variables on the instance (i.e., `self.foo = 3` is a thread-safe operation).

A class-based view is deployed into a URL pattern using the `as_view()` classmethod:

```
urlpatterns = patterns('',
    (r'^view/$', MyView.as_view(size=42)),
)
```

Thread safety with view arguments

Arguments passed to a view are shared between every instance of a view. This means that you shoudn't use a list, dictionary, or any other variable object as an argument to a view. If you did, the actions of one user visiting your view could have an effect on subsequent users visiting the same view.

Any argument passed into `as_view()` will be assigned onto the instance that is used to service a request. Using the previous example, this means that every request on `MyView` is able to use `self.size`.

6.2.7 Base vs Generic views

Base class-based views can be thought of as *parent* views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

Django's generic views are built off of those base views, and were developed as a shortcut for common usage patterns such as displaying the details of an object. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.

Most generic views require the `queryset` key, which is a `QuerySet` instance; see *Making queries* for more information about `QuerySet` objects.

6.3 Clickjacking Protection

The clickjacking middleware and decorators provide easy-to-use protection against clickjacking. This type of attack occurs when a malicious site tricks a user into clicking on a concealed element of another site which they have loaded in a hidden frame or iframe. New in version 1.4: The clickjacking middleware and decorators were added.

6.3.1 An example of clickjacking

Suppose an online store has a page where a logged in user can click "Buy Now" to purchase an item. A user has chosen to stay logged into the store all the time for convenience. An attacker site might create an "I Like Ponies" button on one of their own pages, and load the store's page in a transparent iframe such that the "Buy Now" button is invisibly overlaid on the "I Like Ponies" button. If the user visits the attacker site and clicks "I Like Ponies" he will inadvertently click on the online store's "Buy Now" button and unknowingly purchase the item.

6.3.2 Preventing clickjacking

Modern browsers honor the X-Frame-Options HTTP header that indicates whether or not a resource is allowed to load within a frame or iframe. If the response contains the header with a value of SAMEORIGIN then the browser will only load the resource in a frame if the request originated from the same site. If the header is set to DENY then the browser will block the resource from loading in a frame no matter which site made the request.

Django provides a few simple ways to include this header in responses from your site:

1. A simple middleware that sets the header in all responses.
2. A set of view decorators that can be used to override the middleware or to only set the header for certain views.

6.3.3 How to use it

Setting X-Frame-Options for all responses

To set the same X-Frame-Options value for all responses in your site, add `'django.middleware.clickjacking.XFrameOptionsMiddleware'` to `MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = (
    ...
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ...
)
```

By default, the middleware will set the X-Frame-Options header to SAMEORIGIN for every outgoing `HttpResponse`. If you want DENY instead, set the `X_FRAME_OPTIONS` setting:

```
X_FRAME_OPTIONS = 'DENY'
```

When using the middleware there may be some views where you do **not** want the X-Frame-Options header set. For those cases, you can use a view decorator that tells the middleware not to set the header:

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_exempt

@xframe_options_exempt
def ok_to_load_in_a_frame(request):
    return HttpResponse("This page is safe to load in a frame on any site.")
```

Setting X-Frame-Options per view

To set the X-Frame-Options header on a per view basis, Django provides these decorators:

```
from django.http import HttpResponse
from django.views.decorators.clickjacking import xframe_options_deny
from django.views.decorators.clickjacking import xframe_options_sameorigin

@xframe_options_deny
def view_one(request):
    return HttpResponse("I won't display in any frame!")

@xframe_options_sameorigin
def view_two(request):
    return HttpResponse("Display in a frame if it's from the same origin as me.")
```

Note that you can use the decorators in conjunction with the middleware. Use of a decorator overrides the middleware.

6.3.4 Limitations

The *X-Frame-Options* header will only protect against clickjacking in a modern browser. Older browsers will quietly ignore the header and need other clickjacking prevention techniques.

Browsers that support X-Frame-Options

- Internet Explorer 8+
- Firefox 3.6.9+
- Opera 10.5+
- Safari 4+
- Chrome 4.1+

See also

A complete list of browsers supporting X-Frame-Options.

6.4 contrib packages

Django aims to follow Python's "batteries included" philosophy. It ships with a variety of extra, optional tools that solve common Web-development problems.

This code lives in `django/contrib` in the Django distribution. This document gives a rundown of the packages in `contrib`, along with any dependencies those packages have.

Note

For most of these add-ons – specifically, the add-ons that include either models or template tags – you'll need to add the package name (e.g., `'django.contrib.admin'`) to your `INSTALLED_APPS` setting and re-run `manage.py syncdb`.

6.4.1 The Django admin site

One of the most powerful parts of Django is the automatic admin interface. It reads metadata in your model to provide a powerful and production-ready interface that content producers can immediately use to start adding content to the site. In this document, we discuss how to activate, use and customize Django's admin interface.

Overview

There are seven steps in activating the Django admin site:

1. Add `'django.contrib.admin'` to your `INSTALLED_APPS` setting.
2. The admin has four dependencies - `django.contrib.auth`, `django.contrib.contenttypes`, `django.contrib.messages` and `django.contrib.sessions`. If these applications are not in your `INSTALLED_APPS` list, add them.
3. Add `django.contrib.messages.context_processors.messages` to `TEMPLATE_CONTEXT_PROCESSORS` and `MessageMiddleware` to `MIDDLEWARE_CLASSES`. (These are both active by default, so you only need to do this if you've manually tweaked the settings.)
4. Determine which of your application's models should be editable in the admin interface.
5. For each of those models, optionally create a `ModelAdmin` class that encapsulates the customized admin functionality and options for that particular model.
6. Instantiate an `AdminSite` and tell it about each of your models and `ModelAdmin` classes.
7. Hook the `AdminSite` instance into your URLconf.

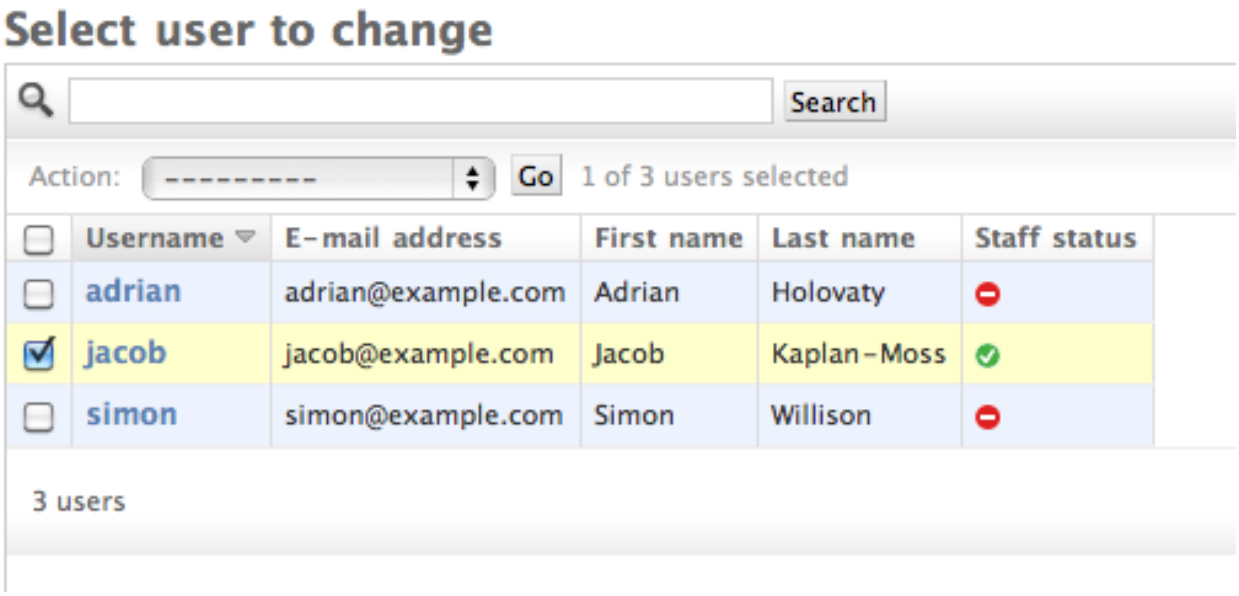
After you've taken these steps, you'll be able to use your Django admin site by visiting the URL you hooked it into (`/admin/`, by default).

Other topics

Admin actions The basic workflow of Django's admin is, in a nutshell, "select an object, then change it." This works well for a majority of use cases. However, if you need to make the same change to many objects at once, this workflow can be quite tedious.

In these cases, Django's admin lets you write and register "actions" – simple functions that get called with a list of objects selected on the change list page.

If you look at any change list in the admin, you'll see this feature in action; Django ships with a "delete selected objects" action available to all models. For example, here's the user module from Django's built-in `django.contrib.auth` app:



Warning: The "delete selected objects" action uses `QuerySet.delete()` for efficiency reasons, which has an important caveat: your model's `delete()` method will not be called.
If you wish to override this behavior, simply write a custom action which accomplishes deletion in your preferred manner – for example, by calling `Model.delete()` for each of the selected items.
For more background on bulk deletion, see the documentation on *object deletion*.

Read on to find out how to add your own actions to this list.

Writing actions The easiest way to explain actions is by example, so let's dive in.

A common use case for admin actions is the bulk updating of a model. Imagine a simple news application with an `Article` model:

```
from django.db import models

STATUS_CHOICES = (
    ('d', 'Draft'),
    ('p', 'Published'),
    ('w', 'Withdrawn'),
)

class Article(models.Model):
    title = models.CharField(max_length=100)
    body = models.TextField()
    status = models.CharField(max_length=1, choices=STATUS_CHOICES)

    def __unicode__(self):
        return self.title
```

A common task we might perform with a model like this is to update an article's status from "draft" to "published". We could easily do this in the admin one article at a time, but if we wanted to bulk-publish a group of articles, it'd be tedious. So, let's write an action that lets us change an article's status to "published."

Writing action functions First, we'll need to write a function that gets called when the action is trigged from the admin. Action functions are just regular functions that take three arguments:

- The current `ModelAdmin`
- An `HttpRequest` representing the current request,
- A `QuerySet` containing the set of objects selected by the user.

Our publish-these-articles function won't need the `ModelAdmin` or the request object, but we will use the queryset:

```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
```

Note: For the best performance, we're using the queryset's *update method*. Other types of actions might need to deal with each object individually; in these cases we'd just iterate over the queryset:

```
for obj in queryset:
    do_something_with(obj)
```

That's actually all there is to writing an action! However, we'll take one more optional-but-useful step and give the action a "nice" title in the admin. By default, this action would appear in the action list as "Make published" – the function name, with underscores replaced by spaces. That's fine, but we can provide a better, more human-friendly name by giving the `make_published` function a `short_description` attribute:

```
def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"
```

Note: This might look familiar; the admin's `list_display` option uses the same technique to provide human-readable descriptions for callback functions registered there, too.

Adding actions to the `ModelAdmin` Next, we'll need to inform our `ModelAdmin` of the action. This works just like any other configuration option. So, the complete `admin.py` with the action and its registration would look like:

```
from django.contrib import admin
from myapp.models import Article

def make_published(modeladmin, request, queryset):
    queryset.update(status='p')
make_published.short_description = "Mark selected stories as published"

class ArticleAdmin(admin.ModelAdmin):
    list_display = ['title', 'status']
    ordering = ['title']
    actions = [make_published]

admin.site.register(Article, ArticleAdmin)
```

That code will give us an admin change list that looks something like this:

That's really all there is to it! If you're itching to write your own actions, you now know enough to get started. The rest of this document just covers more advanced techniques.

Advanced action techniques There's a couple of extra options and possibilities you can exploit for more advanced options.

Actions as `ModelAdmin` methods The example above shows the `make_published` action defined as a simple function. That's perfectly fine, but it's not perfect from a code design point of view: since the action is tightly coupled to the `Article` object, it makes sense to hook the action to the `ArticleAdmin` object itself.

That's easy enough to do:

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    actions = ['make_published']

    def make_published(self, request, queryset):
        queryset.update(status='p')
    make_published.short_description = "Mark selected stories as published"
```

Notice first that we've moved `make_published` into a method and renamed the *modeladmin* parameter to *self*, and second that we've now put the string `'make_published'` in `actions` instead of a direct function reference. This tells the `ModelAdmin` to look up the action as a method.

Defining actions as methods gives the action more straightforward, idiomatic access to the `ModelAdmin` itself, allowing the action to call any of the methods provided by the admin. For example, we can use `self` to flash a message to the user informing her that the action was successful:

```
class ArticleAdmin(admin.ModelAdmin):
    ...

    def make_published(self, request, queryset):
        rows_updated = queryset.update(status='p')
        if rows_updated == 1:
            message_bit = "1 story was"
        else:
```

```
            message_bit = "%s stories were" % rows_updated
        self.message_user(request, "%s successfully marked as published." % message_bit)
```

This make the action match what the admin itself does after successfully performing an action:

Actions that provide intermediate pages By default, after an action is performed the user is simply redirected back to the original change list page. However, some actions, especially more complex ones, will need to return intermediate pages. For example, the built-in delete action asks for confirmation before deleting the selected objects.

To provide an intermediary page, simply return an `HttpResponse` (or subclass) from your action. For example, you might write a simple export function that uses Django's *serialization functions* to dump some selected objects as JSON:

```
from django.http import HttpResponse
from django.core import serializers

def export_as_json(modeladmin, request, queryset):
    response = HttpResponse(mimetype="text/javascript")
    serializers.serialize("json", queryset, stream=response)
    return response
```

Generally, something like the above isn't considered a great idea. Most of the time, the best practice will be to return an `HttpResponseRedirect` and redirect the user to a view you've written, passing the list of selected objects in the GET query string. This allows you to provide complex interaction logic on the intermediary pages. For example, if you wanted to provide a more complete export function, you'd want to let the user choose a format, and possibly a list of fields to include in the export. The best thing to do would be to write a small action that simply redirects to your custom export view:

```
from django.contrib import admin
from django.contrib.contenttypes.models import ContentType
from django.http import HttpResponseRedirect

def export_selected_objects(modeladmin, request, queryset):
    selected = request.POST.getlist(admin.ACTION_CHECKBOX_NAME)
    ct = ContentType.objects.get_for_model(queryset.model)
    return HttpResponseRedirect("/export/?ct=%s&ids=%s" % (ct.pk, ",".join(selected)))
```

As you can see, the action is the simple part; all the complex logic would belong in your export view. This would need to deal with objects of any type, hence the business with the `ContentType`.

Writing this view is left as an exercise to the reader.

Making actions available site-wide

`AdminSite.`**`add_action`**`(`*action*`[`, *name*`])`

Some actions are best if they're made available to *any* object in the admin site – the export action defined above would be a good candidate. You can make an action globally available using `AdminSite.add_action()`. For example:

```
from django.contrib import admin

admin.site.add_action(export_selected_objects)
```

This makes the *export_selected_objects* action globally available as an action named *"export_selected_objects"*. You can explicitly give the action a name – good if you later want to programatically *remove the action* – by passing a second argument to `AdminSite.add_action()`:

```
admin.site.add_action(export_selected_objects, 'export_selected')
```

Disabling actions Sometimes you need to disable certain actions – especially those *registered site-wide* – for particular objects. There's a few ways you can disable actions:

Disabling a site-wide action

`AdminSite.`**`disable_action`**`(`*name*`)`

If you need to disable a *site-wide action* you can call `AdminSite.disable_action()`.

For example, you can use this method to remove the built-in "delete selected objects" action:

```
admin.site.disable_action('delete_selected')
```

Once you've done the above, that action will no longer be available site-wide.

If, however, you need to re-enable a globally-disabled action for one particular model, simply list it explicitly in your `ModelAdmin.actions` list:

```
# Globally disable delete selected
admin.site.disable_action('delete_selected')

# This ModelAdmin will not have delete_selected available
class SomeModelAdmin(admin.ModelAdmin):
    actions = ['some_other_action']
    ...

# This one will
class AnotherModelAdmin(admin.ModelAdmin):
    actions = ['delete_selected', 'a_third_action']
    ...
```

Disabling all actions for a particular `ModelAdmin` If you want *no* bulk actions available for a given `ModelAdmin`, simply set `ModelAdmin.actions` to `None`:

```
class MyModelAdmin(admin.ModelAdmin):
    actions = None
```

This tells the `ModelAdmin` to not display or allow any actions, including any *site-wide actions*.

Conditionally enabling or disabling actions

`ModelAdmin.get_actions(request)`

Finally, you can conditionally enable or disable actions on a per-request (and hence per-user basis) by overriding `ModelAdmin.get_actions()`.

This returns a dictionary of actions allowed. The keys are action names, and the values are `(function, name, short_description)` tuples.

Most of the time you'll use this method to conditionally remove actions from the list gathered by the superclass. For example, if I only wanted users whose names begin with 'J' to be able to delete objects in bulk, I could do the following:

```
class MyModelAdmin(admin.ModelAdmin):
    ...

    def get_actions(self, request):
        actions = super(MyModelAdmin, self).get_actions(request)
        if request.user.username[0].upper() != 'J':
            if 'delete_selected' in actions:
                del actions['delete_selected']
        return actions
```

The Django admin documentation generator Django's `admindocs` app pulls documentation from the docstrings of models, views, template tags, and template filters for any app in `INSTALLED_APPS` and makes that documentation available from the `Django admin`.

In addition to providing offline documentation for all template tags and template filters that ship with Django, you may utilize admindocs to quickly document your own code.

Overview To activate the `admindocs`, you will need to do the following:

- Add `django.contrib.admindocs` to your `INSTALLED_APPS`.
- Add `(r'^admin/doc/', include('django.contrib.admindocs.urls'))` to your `urlpatterns`. Make sure it's included *before* the `r'^admin/'` entry, so that requests to `/admin/doc/` don't get handled by the latter entry.
- Install the docutils Python module (http://docutils.sf.net/).
- **Optional:** Linking to templates requires the `ADMIN_FOR` setting to be configured.
- **Optional:** Using the admindocs bookmarklets requires the `XViewMiddleware` to be installed.

Once those steps are complete, you can start browsing the documentation by going to your admin interface and clicking the "Documentation" link in the upper right of the page.

Documentation helpers The following special markup can be used in your docstrings to easily create hyperlinks to other components:

Django Component	reStructuredText roles
Models	`:model:`appname.ModelName``
Views	`:view:`appname.view_name``
Template tags	`:tag:`tagname``
Template filters	`:filter:`filtername``
Templates	`:template:`path/to/template.html``

Model reference The **models** section of the `admindocs` page describes each model in the system along with all the fields and methods available on it. Relationships to other models appear as hyperlinks. Descriptions are pulled from `help_text` attributes on fields or from docstrings on model methods.

A model with useful documentation might look like this:

```
class BlogEntry(models.Model):
    """
    Stores a single blog entry, related to :model:`blog.Blog` and
    :model:`auth.User`.

    """
    slug = models.SlugField(help_text="A short label, generally used in URLs.")
    author = models.ForeignKey(User)
    blog = models.ForeignKey(Blog)
    ...

    def publish(self):
        """Makes the blog entry live on the site."""
        ...
```

View reference Each URL in your site has a separate entry in the `admindocs` page, and clicking on a given URL will show you the corresponding view. Helpful things you can document in your view function docstrings include:

- A short description of what the view does.
- The **context**, or a list of variables available in the view's template.
- The name of the template or templates that are used for that view.

For example:

```
from myapp.models import MyModel

def my_view(request, slug):
    """
    Display an individual :model:`myapp.MyModel`.

    **Context**

    ``RequestContext``

    ``mymodel``
        An instance of :model:`myapp.MyModel`.

    **Template:**

    :template:`myapp/my_template.html`

    """
    return render_to_response('myapp/my_template.html', {
        'mymodel': MyModel.objects.get(slug=slug)
    }, context_instance=RequestContext(request))
```

Template tags and filters reference The **tags** and **filters** `admindocs` sections describe all the tags and filters that come with Django (in fact, the *built-in tag reference* and *built-in filter reference* documentation come directly from those pages). Any tags or filters that you create or are added by a third-party app will show up in these sections as well.

Template reference While `admindocs` does not include a place to document templates by themselves, if you use the `:template:`path/to/template.html`` syntax in a docstring the resulting page will verify the path of that template with Django's *template loaders*. This can be a handy way to check if the specified template exists and to show where on the filesystem that template is stored.

Included Bookmarklets Several useful bookmarklets are available from the `admindocs` page:

Documentation for this page Jumps you from any page to the documentation for the view that generates that page.

Show object ID Shows the content-type and unique ID for pages that represent a single object.

Edit this object Jumps to the admin page for pages that represent a single object.

Using these bookmarklets requires that you are either logged into the `Django admin` as a `User` with `is_staff` set to *True*, or that the `django.middleware.doc` middleware and `XViewMiddleware` are installed and you are accessing the site from an IP address listed in `INTERNAL_IPS`.

See Also:

For information about serving the static files (images, JavaScript, and CSS) associated with the admin in production, see *Serving files*.

`ModelAdmin` objects

class `ModelAdmin`

The `ModelAdmin` class is the representation of a model in the admin interface. These are stored in a file named `admin.py` in your application. Let's take a look at a very simple example of the `ModelAdmin`:

```
from django.contrib import admin
from myproject.myapp.models import Author

class AuthorAdmin(admin.ModelAdmin):
    pass
admin.site.register(Author, AuthorAdmin)
```

Do you need a `ModelAdmin` object at all?

In the preceding example, the `ModelAdmin` class doesn't define any custom values (yet). As a result, the default admin interface will be provided. If you are happy with the default admin interface, you don't need to define a `ModelAdmin` object at all – you can register the model class without providing a `ModelAdmin` description. The preceding example could be simplified to:

```
from django.contrib import admin
from myproject.myapp.models import Author

admin.site.register(Author)
```

`ModelAdmin` options

The `ModelAdmin` is very flexible. It has several options for dealing with customizing the interface. All options are defined on the `ModelAdmin` subclass:

```
class AuthorAdmin(admin.ModelAdmin):
    date_hierarchy = 'pub_date'
```

ModelAdmin.**actions**
A list of actions to make available on the change list page. See *Admin actions* for details.

ModelAdmin.**actions_on_top**

ModelAdmin.**actions_on_bottom**
Controls where on the page the actions bar appears. By default, the admin changelist displays actions at the top of the page (`actions_on_top = True; actions_on_bottom = False`).

ModelAdmin.**actions_selection_counter**
Controls whether a selection counter is displayed next to the action dropdown. By default, the admin changelist will display it (`actions_selection_counter = True`).

ModelAdmin.**date_hierarchy**
Set `date_hierarchy` to the name of a `DateField` or `DateTimeField` in your model, and the change list page will include a date-based drilldown navigation by that field.

Example:

```
date_hierarchy = 'pub_date'
```

New in version 1.3: *Please see the release notes* This will intelligently populate itself based on available data, e.g. if all the dates are in one month, it'll show the day-level drill-down only.

ModelAdmin.**exclude**
This attribute, if given, should be a list of field names to exclude from the form.

For example, let's consider the following model:

```
class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3)
    birth_date = models.DateField(blank=True, null=True)
```

If you want a form for the `Author` model that includes only the `name` and `title` fields, you would specify `fields` or `exclude` like this:

```
class AuthorAdmin(admin.ModelAdmin):
    fields = ('name', 'title')

class AuthorAdmin(admin.ModelAdmin):
    exclude = ('birth_date',)
```

Since the Author model only has three fields, `name`, `title`, and `birth_date`, the forms resulting from the above declarations will contain exactly the same fields.

ModelAdmin.**fields**
If you need to achieve simple changes in the layout of fields in the forms of the "add" and "change" pages like only showing a subset of the available fields, modifying their order or grouping them in rows you can use the `fields` option (for more complex layout needs see the `fieldsets` option described in the next section). For example, you could define a simpler version of the admin form for the `django.contrib.flatpages.FlatPage` model as follows:

```
class FlatPageAdmin(admin.ModelAdmin):
    fields = ('url', 'title', 'content')
```

In the above example, only the fields `url`, `title` and `content` will be displayed, sequentially, in the form. `fields` can contain values defined in `ModelAdmin.readonly_fields` to be displayed as read-only. New in version 1.4: *Please see the release notes* To display multiple fields on the same line, wrap those fields in their own tuple. In this example, the `url` and `title` fields will display on the same line and the `content` field will be displayed below them in its own line:

```
class FlatPageAdmin(admin.ModelAdmin):
    fields = (('url', 'title'), 'content')
```

Note

This `fields` option should not be confused with the `fields` dictionary key that is within the `fields` option, as described in the next section.

If neither `fields` nor `fieldsets` options are present, Django will default to displaying each field that isn't an `AutoField` and has `editable=True`, in a single fieldset, in the same order as the fields are defined in the model.

`ModelAdmin.`**`fieldsets`**

Set `fieldsets` to control the layout of admin "add" and "change" pages.

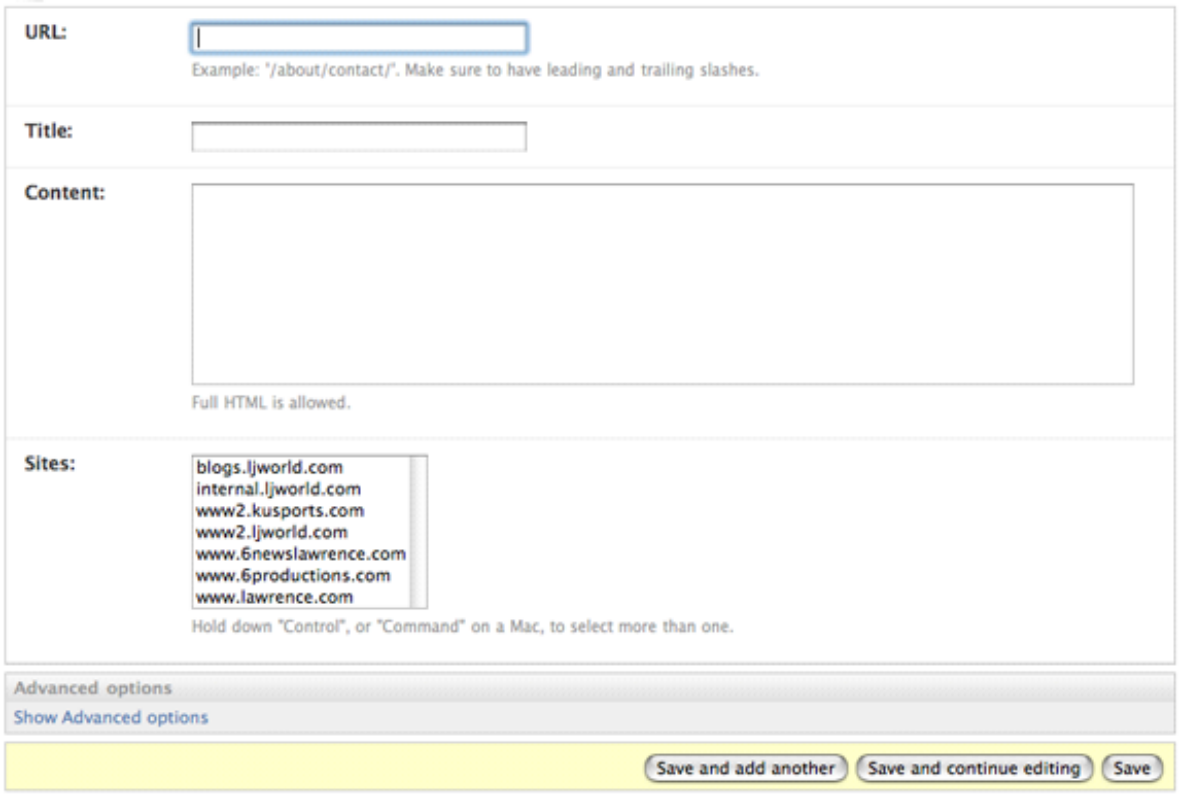
`fieldsets` is a list of two-tuples, in which each two-tuple represents a `<fieldset>` on the admin form page. (A `<fieldset>` is a "section" of the form.)

The two-tuples are in the format `(name, field_options)`, where `name` is a string representing the title of the fieldset and `field_options` is a dictionary of information about the fieldset, including a list of fields to be displayed in it.

A full example, taken from the `django.contrib.flatpages.FlatPage` model:

```
class FlatPageAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('url', 'title', 'content', 'sites')
        }),
        ('Advanced options', {
            'classes': ('collapse',),
            'fields': ('enable_comments', 'registration_required', 'template_name')
        }),
    )
```

This results in an admin page that looks like:



If neither `fieldsets` nor `fields` options are present, Django will default to displaying each field that isn't an `AutoField` and has `editable=True`, in a single fieldset, in the same order as the fields are defined in the model.

The `field_options` dictionary can have the following keys:

- **`fields`** A tuple of field names to display in this fieldset. This key is required.

  Example:

  ```
  {
  'fields': ('first_name', 'last_name', 'address', 'city', 'state'),
  }
  ```

  Just like with the `fields` option, to display multiple fields on the same line, wrap those fields in their own tuple. In this example, the `first_name` and `last_name` fields will display on the same line:

  ```
  {
  'fields': (('first_name', 'last_name'), 'address', 'city', 'state'),
  }
  ```

  `fields` can contain values defined in `readonly_fields` to be displayed as read-only.

- **`classes`** A list containing extra CSS classes to apply to the fieldset.

  Example:

  ```
  {
  'classes': ['wide', 'extrapretty'],
  }
  ```

Two useful classes defined by the default admin site stylesheet are `collapse` and `wide`. Fieldsets with the `collapse` style will be initially collapsed in the admin and replaced with a small "click to expand" link. Fieldsets with the `wide` style will be given extra horizontal space.

- **`description`** A string of optional extra text to be displayed at the top of each fieldset, under the heading of the fieldset.

  Note that this value is *not* HTML-escaped when it's displayed in the admin interface. This lets you include HTML if you so desire. Alternatively you can use plain text and `django.utils.html.escape()` to escape any HTML special characters.

`ModelAdmin.`**`filter_horizontal`**

By default, a `ManyToManyField` is displayed in the admin site with a `<select multiple>`. However, multiple-select boxes can be difficult to use when selecting many items. Adding a `ManyToManyField` to this list will instead use a nifty unobtrusive JavaScript "filter" interface that allows searching within the options. The unselected and selected options appear in two boxes side by side. See `filter_vertical` to use a vertical interface.

`ModelAdmin.`**`filter_vertical`**

Same as `filter_horizontal`, but uses a vertical display of the filter interface with the box of unselected options appearing above the box of selected options.

`ModelAdmin.`**`form`**

By default a `ModelForm` is dynamically created for your model. It is used to create the form presented on both the add/change pages. You can easily provide your own `ModelForm` to override any default form behavior on the add/change pages.

For an example see the section Adding custom validation to the admin.

Note

If your `ModelForm` and `ModelAdmin` both define an `exclude` option then `ModelAdmin` takes precedence:

```
class PersonForm(forms.ModelForm):

    class Meta:
        model = Person
        exclude = ['name']

class PersonAdmin(admin.ModelAdmin):
    exclude = ['age']
    form = PersonForm
```

In the above example, the "age" field will be excluded but the "name" field will be included in the generated form.

`ModelAdmin.`**`formfield_overrides`**

This provides a quick-and-dirty way to override some of the `Field` options for use in the admin. `formfield_overrides` is a dictionary mapping a field class to a dict of arguments to pass to the field at construction time.

Since that's a bit abstract, let's look at a concrete example. The most common use of `formfield_overrides` is to add a custom widget for a certain type of field. So, imagine we've written a `RichTextEditorWidget` that we'd like to use for large text fields instead of the default `<textarea>`. Here's how we'd do that:

```
from django.db import models
from django.contrib import admin
```

```
# Import our custom widget and our model from where they're defined
from myapp.widgets import RichTextEditorWidget
from myapp.models import MyModel

class MyModelAdmin(admin.ModelAdmin):
    formfield_overrides = {
        models.TextField: {'widget': RichTextEditorWidget},
    }
```

Note that the key in the dictionary is the actual field class, *not* a string. The value is another dictionary; these arguments will be passed to `__init__()`. See *The Forms API* for details.

> **Warning:** If you want to use a custom widget with a relation field (i.e. `ForeignKey` or `ManyToManyField`), make sure you haven't included that field's name in `raw_id_fields` or `radio_fields`.
> `formfield_overrides` won't let you change the widget on relation fields that have `raw_id_fields` or `radio_fields` set. That's because `raw_id_fields` and `radio_fields` imply custom widgets of their own.

`ModelAdmin.`**`inlines`**

See `InlineModelAdmin` objects below.

`ModelAdmin.`**`list_display`**

Set `list_display` to control which fields are displayed on the change list page of the admin.

Example:

```
list_display = ('first_name', 'last_name')
```

If you don't set `list_display`, the admin site will display a single column that displays the `__unicode__()` representation of each object.

You have four possible values that can be used in `list_display`:

- A field of the model. For example:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name')
```

- A callable that accepts one parameter for the model instance. For example:

```
def upper_case_name(obj):
    return ("%s %s" % (obj.first_name, obj.last_name)).upper()
upper_case_name.short_description = 'Name'

class PersonAdmin(admin.ModelAdmin):
    list_display = (upper_case_name,)
```

- A string representing an attribute on the `ModelAdmin`. This behaves same as the callable. For example:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('upper_case_name',)

    def upper_case_name(self, obj):
      return ("%s %s" % (obj.first_name, obj.last_name)).upper()
    upper_case_name.short_description = 'Name'
```

- A string representing an attribute on the model. This behaves almost the same as the callable, but `self` in this context is the model instance. Here's a full model example:

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    birthday = models.DateField()

    def decade_born_in(self):
        return self.birthday.strftime('%Y')[:3] + "0's"
    decade_born_in.short_description = 'Birth decade'

class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'decade_born_in')
```

A few special cases to note about `list_display`:

- If the field is a `ForeignKey`, Django will display the `__unicode__()` of the related object.
- `ManyToManyField` fields aren't supported, because that would entail executing a separate SQL statement for each row in the table. If you want to do this nonetheless, give your model a custom method, and add that method's name to `list_display`. (See below for more on custom methods in `list_display`.)
- If the field is a `BooleanField` or `NullBooleanField`, Django will display a pretty "on" or "off" icon instead of `True` or `False`.
- If the string given is a method of the model, `ModelAdmin` or a callable, Django will HTML-escape the output by default. If you'd rather not escape the output of the method, give the method an `allow_tags` attribute whose value is `True`.

  Here's a full example model:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_name(self):
        return '<span style="color: #%s;">%s %s</span>' % (self.color_code, self.first_name,
    colored_name.allow_tags = True

class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'colored_name')
```

- If the string given is a method of the model, `ModelAdmin` or a callable that returns True or False Django will display a pretty "on" or "off" icon if you give the method a `boolean` attribute whose value is `True`.

  Here's a full example model:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    birthday = models.DateField()

    def born_in_fifties(self):
        return self.birthday.strftime('%Y')[:3] == '195'
    born_in_fifties.boolean = True

class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'born_in_fifties')
```

- The `__str__()` and `__unicode__()` methods are just as valid in `list_display` as any other model method, so it's perfectly OK to do this:

```
list_display = ('__unicode__', 'some_other_field')
```

- Usually, elements of `list_display` that aren't actual database fields can't be used in sorting (because Django does all the sorting at the database level).

  However, if an element of `list_display` represents a certain database field, you can indicate this fact by setting the `admin_order_field` attribute of the item.

  For example:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_first_name(self):
        return '<span style="color: #%s;">%s</span>' % (self.color_code, self.first_name)
    colored_first_name.allow_tags = True
    colored_first_name.admin_order_field = 'first_name'

class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'colored_first_name')
```

  The above will tell Django to order by the `first_name` field when trying to sort by `colored_first_name` in the admin.

`ModelAdmin.`**`list_display_links`**

Set `list_display_links` to control which fields in `list_display` should be linked to the "change" page for an object.

By default, the change list page will link the first column – the first field specified in `list_display` – to the change page for each item. But `list_display_links` lets you change which columns are linked. Set `list_display_links` to a list or tuple of fields (in the same format as `list_display`) to link.

`list_display_links` can specify one or many fields. As long as the fields appear in `list_display`, Django doesn't care how many (or how few) fields are linked. The only requirement is: If you want to use `list_display_links`, you must define `list_display`.

In this example, the `first_name` and `last_name` fields will be linked on the change list page:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'birthday')
    list_display_links = ('first_name', 'last_name')
```

`ModelAdmin.`**`list_editable`**

Set `list_editable` to a list of field names on the model which will allow editing on the change list page. That is, fields listed in `list_editable` will be displayed as form widgets on the change list page, allowing users to edit and save multiple rows at once.

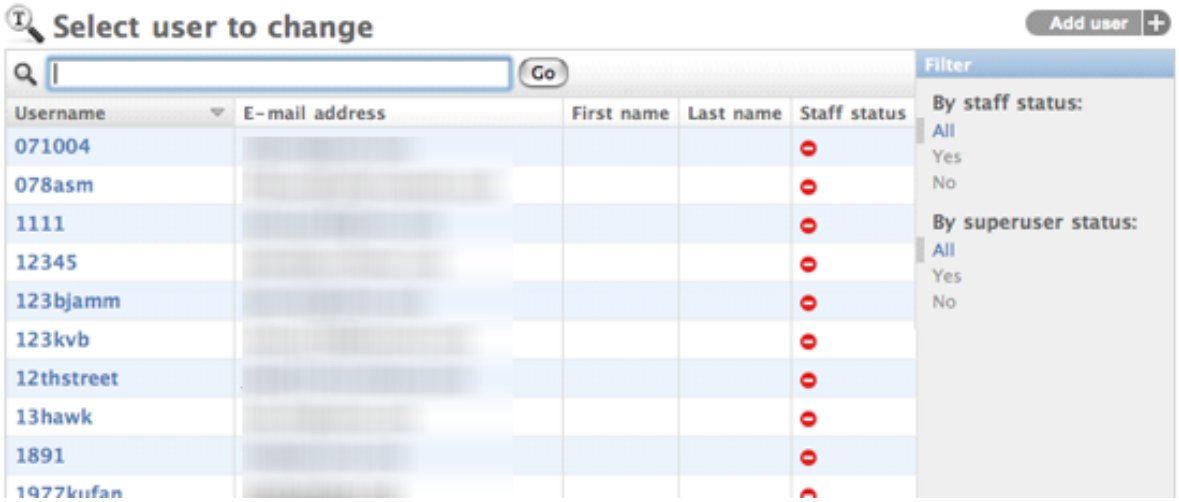
Note: `list_editable` interacts with a couple of other options in particular ways; you should note the following rules:

- Any field in `list_editable` must also be in `list_display`. You can't edit a field that's not displayed!
- The same field can't be listed in both `list_editable` and `list_display_links` – a field can't be both a form and a link.

You'll get a validation error if either of these rules are broken.

ModelAdmin.**list_filter**

Changed in version 1.4: *Please see the release notes* Set `list_filter` to activate filters in the right sidebar of the change list page of the admin, as illustrated in the following screenshot:



`list_filter` should be a list of elements, where each element should be of one of the following types:

- a field name, where the specified field should be either a `BooleanField`, `CharField`, `DateField`, `DateTimeField`, `IntegerField`, `ForeignKey` or `ManyToManyField`, for example:

```
class PersonAdmin(ModelAdmin):
    list_filter = ('is_staff', 'company')
```

New in version 1.3: *Please see the release notes* Field names in `list_filter` can also span relations using the __ lookup, for example:

```
class PersonAdmin(UserAdmin):
    list_filter = ('company__name',)
```

- a class inheriting from `django.contrib.admin.SimpleListFilter`, which you need to provide the `title` and `parameter_name` attributes to and override the `lookups` and `queryset` methods, e.g.:

```
from datetime import date

from django.utils.translation import ugettext_lazy as _
from django.contrib.admin import SimpleListFilter

class DecadeBornListFilter(SimpleListFilter):
    # Human-readable title which will be displayed in the
    # right admin sidebar just above the filter options.
    title = _('decade born')

    # Parameter for the filter that will be used in the URL query.
    parameter_name = 'decade'

    def lookups(self, request, model_admin):
        """
        Returns a list of tuples. The first element in each
        tuple is the coded value for the option that will
        appear in the URL query. The second element is the
        human-readable name for the option that will appear
        in the right sidebar.
```

```
        """
        return (
            ('80s', _('in the eighties')),
            ('90s', _('in the nineties')),
        )

    def queryset(self, request, queryset):
        """
        Returns the filtered queryset based on the value
        provided in the query string and retrievable via
        `self.value()`.
        """
        # Compare the requested value (either '80s' or 'other')
        # to decide how to filter the queryset.
        if self.value() == '80s':
            return queryset.filter(birthday__gte=date(1980, 1, 1),
                                   birthday__lte=date(1989, 12, 31))
        if self.value() == '90s':
            return queryset.filter(birthday__gte=date(1990, 1, 1),
                                   birthday__lte=date(1999, 12, 31))

class PersonAdmin(ModelAdmin):
    list_filter = (DecadeBornListFilter,)
```

Note: As a convenience, the `HttpRequest` object is passed to the `lookups` and `queryset` methods, for example:

```
class AuthDecadeBornListFilter(DecadeBornListFilter):

    def lookups(self, request, model_admin):
        if request.user.is_superuser:
            return super(AuthDecadeBornListFilter,
                self).lookups(request, model_admin)

    def queryset(self, request, queryset):
        if request.user.is_superuser:
            return super(AuthDecadeBornListFilter,
                self).queryset(request, queryset)
```

Also as a convenience, the `ModelAdmin` object is passed to the `lookups` method, for example if you want to base the lookups on the available data:

```
class AdvancedDecadeBornListFilter(DecadeBornListFilter):

    def lookups(self, request, model_admin):
        """
        Only show the lookups if there actually is
        anyone born in the corresponding decades.
        """
        qs = model_admin.queryset(request)
        if qs.filter(birthday__gte=date(1980, 1, 1),
                     birthday__lte=date(1989, 12, 31)).exists():
            yield ('80s', _('in the eighties'))
        if qs.filter(birthday__gte=date(1990, 1, 1),
                     birthday__lte=date(1999, 12, 31)).exists():
            yield ('90s', _('in the nineties'))
```

- a tuple, where the first element is a field name and the second element is a class inheriting from `django.contrib.admin.FieldListFilter`, for example:

```
from django.contrib.admin import BooleanFieldListFilter

class PersonAdmin(ModelAdmin):
    list_filter = (
        ('is_staff', BooleanFieldListFilter),
    )
```

Note: The `FieldListFilter` API is considered internal and might be changed.

New in version 1.4: *Please see the release notes* It is possible to specify a custom template for rendering a list filter:

```
class FilterWithCustomTemplate(SimpleListFilter):
    template = "custom_template.html"
```

See the default template provided by django (`admin/filter.html`) for a concrete example.

`ModelAdmin.`**`list_max_show_all`**

New in version 1.4: *Please see the release notes* Set `list_max_show_all` to control how many items can appear on a "Show all" admin change list page. The admin will display a "Show all" link on the change list only if the total result count is less than or equal to this setting. By default, this is set to `200`.

`ModelAdmin.`**`list_per_page`**

Set `list_per_page` to control how many items appear on each paginated admin change list page. By default, this is set to `100`.

`ModelAdmin.`**`list_select_related`**

Set `list_select_related` to tell Django to use `select_related()` in retrieving the list of objects on the admin change list page. This can save you a bunch of database queries.

The value should be either `True` or `False`. Default is `False`.

Note that Django will use `select_related()`, regardless of this setting if one of the `list_display` fields is a `ForeignKey`.

`ModelAdmin.`**`ordering`**

Set `ordering` to specify how lists of objects should be ordered in the Django admin views. This should be a list or tuple in the same format as a model's `ordering` parameter.

If this isn't provided, the Django admin will use the model's default ordering. New in version 1.4: *Please see the release notes* If you need to specify a dynamic order (for example depending on user or language) you can implement a `get_ordering()` method. Changed in version 1.4: *Please see the release notes* Django honors all elements in the list/tuple; before 1.4, only the first was respected.

`ModelAdmin.`**`paginator`**

New in version 1.3: *Please see the release notes* The paginator class to be used for pagination. By default, `django.core.paginator.Paginator` is used. If the custom paginator class doesn't have the same constructor interface as `django.core.paginator.Paginator`, you will also need to provide an implementation for `ModelAdmin.get_paginator()`.

`ModelAdmin.`**`prepopulated_fields`**

Set `prepopulated_fields` to a dictionary mapping field names to the fields it should prepopulate from:

```
class ArticleAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
```

When set, the given fields will use a bit of JavaScript to populate from the fields assigned. The main use for this functionality is to automatically generate the value for `SlugField` fields from one or more other fields. The generated value is produced by concatenating the values of the source fields, and then by transforming that result into a valid slug (e.g. substituting dashes for spaces).

`prepopulated_fields` doesn't accept `DateTimeField`, `ForeignKey`, nor `ManyToManyField` fields.

`ModelAdmin.`**`radio_fields`**

By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey` or have `choices` set. If a field is present in `radio_fields`, Django will use a radio-button interface instead. Assuming `group` is a `ForeignKey` on the `Person` model:

```
class PersonAdmin(admin.ModelAdmin):
    radio_fields = {"group": admin.VERTICAL}
```

You have the choice of using `HORIZONTAL` or `VERTICAL` from the `django.contrib.admin` module.

Don't include a field in `radio_fields` unless it's a `ForeignKey` or has `choices` set.

`ModelAdmin.`**`raw_id_fields`**

By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey`. Sometimes you don't want to incur the overhead of having to select all the related instances to display in the drop-down.

`raw_id_fields` is a list of fields you would like to change into an `Input` widget for either a `ForeignKey` or `ManyToManyField`:

```
class ArticleAdmin(admin.ModelAdmin):
    raw_id_fields = ("newspaper",)
```

`ModelAdmin.`**`readonly_fields`**

By default the admin shows all fields as editable. Any fields in this option (which should be a `list` or `tuple`) will display its data as-is and non-editable. This option behaves nearly identical to `ModelAdmin.list_display`. Usage is the same, however, when you specify `ModelAdmin.fields` or `ModelAdmin.fieldsets` the read-only fields must be present to be shown (they are ignored otherwise).

If `readonly_fields` is used without defining explicit ordering through `ModelAdmin.fields` or `ModelAdmin.fieldsets` they will be added last after all editable fields.

`ModelAdmin.`**`save_as`**

Set `save_as` to enable a "save as" feature on admin change forms.

Normally, objects have three save options: "Save", "Save and continue editing" and "Save and add another". If `save_as` is `True`, "Save and add another" will be replaced by a "Save as" button.

"Save as" means the object will be saved as a new object (with a new ID), rather than the old object.

By default, `save_as` is set to `False`.

`ModelAdmin.`**`save_on_top`**

Set `save_on_top` to add save buttons across the top of your admin change forms.

Normally, the save buttons appear only at the bottom of the forms. If you set `save_on_top`, the buttons will appear both on the top and the bottom.

By default, `save_on_top` is set to `False`.

`ModelAdmin.`**`search_fields`**

Set `search_fields` to enable a search box on the admin change list page. This should be set to a list of field names that will be searched whenever somebody submits a search query in that text box.

These fields should be some kind of text field, such as `CharField` or `TextField`. You can also perform a related lookup on a `ForeignKey` or `ManyToManyField` with the lookup API "follow" notation:

```
search_fields = ['foreign_key__related_fieldname']
```

For example, if you have a blog entry with an author, the following definition would enable search blog entries by the email address of the author:

```
search_fields = ['user__email']
```

When somebody does a search in the admin search box, Django splits the search query into words and returns all objects that contain each of the words, case insensitive, where each word must be in at least one of `search_fields`. For example, if `search_fields` is set to `['first_name', 'last_name']` and a user searches for `john lennon`, Django will do the equivalent of this SQL `WHERE` clause:

```
WHERE (first_name ILIKE '%john%' OR last_name ILIKE '%john%')
AND (first_name ILIKE '%lennon%' OR last_name ILIKE '%lennon%')
```

For faster and/or more restrictive searches, prefix the field name with an operator:

^ Matches the beginning of the field. For example, if `search_fields` is set to `['^first_name', '^last_name']` and a user searches for `john lennon`, Django will do the equivalent of this SQL `WHERE` clause:

```
WHERE (first_name ILIKE 'john%' OR last_name ILIKE 'john%')
AND (first_name ILIKE 'lennon%' OR last_name ILIKE 'lennon%')
```

This query is more efficient than the normal `'%john%'` query, because the database only needs to check the beginning of a column's data, rather than seeking through the entire column's data. Plus, if the column has an index on it, some databases may be able to use the index for this query, even though it's a `LIKE` query.

= Matches exactly, case-insensitive. For example, if `search_fields` is set to `['=first_name', '=last_name']` and a user searches for `john lennon`, Django will do the equivalent of this SQL `WHERE` clause:

```
WHERE (first_name ILIKE 'john' OR last_name ILIKE 'john')
AND (first_name ILIKE 'lennon' OR last_name ILIKE 'lennon')
```

Note that the query input is split by spaces, so, following this example, it's currently not possible to search for all records in which `first_name` is exactly `'john winston'` (containing a space).

@ Performs a full-text match. This is like the default search method but uses an index. Currently this is only available for MySQL.

Custom template options The Overriding Admin Templates section describes how to override or extend the default admin templates. Use the following options to override the default templates used by the `ModelAdmin` views:

`ModelAdmin.`**`add_form_template`**
Path to a custom template, used by `add_view()`.

`ModelAdmin.`**`change_form_template`**
Path to a custom template, used by `change_view()`.

`ModelAdmin.`**`change_list_template`**
Path to a custom template, used by `changelist_view()`.

`ModelAdmin.`**`delete_confirmation_template`**
Path to a custom template, used by `delete_view()` for displaying a confirmation page when deleting one or more objects.

ModelAdmin.**delete_selected_confirmation_template**
Path to a custom template, used by the `delete_selected()` action method for displaying a confirmation page when deleting one or more objects. See the *actions documentation*.

ModelAdmin.**object_history_template**
Path to a custom template, used by `history_view()`.

`ModelAdmin` methods

Warning: `ModelAdmin.save_model()` and `ModelAdmin.delete_model()` must save/delete the object, they are not for veto purposes, rather they allow you to perform extra operations.

ModelAdmin.**save_model**(*self*, *request*, *obj*, *form*, *change*)
The `save_model` method is given the `HttpRequest`, a model instance, a `ModelForm` instance and a boolean value based on whether it is adding or changing the object. Here you can do any pre- or post-save operations.

For example to attach `request.user` to the object prior to saving:

```
class ArticleAdmin(admin.ModelAdmin):
    def save_model(self, request, obj, form, change):
        obj.user = request.user
        obj.save()
```

ModelAdmin.**delete_model**(*self*, *request*, *obj*)
New in version 1.3: *Please see the release notes* The `delete_model` method is given the `HttpRequest` and a model instance. Use this method to do pre- or post-delete operations.

ModelAdmin.**save_formset**(*self*, *request*, *form*, *formset*, *change*)
The `save_formset` method is given the `HttpRequest`, the parent `ModelForm` instance and a boolean value based on whether it is adding or changing the parent object.

For example to attach `request.user` to each changed formset model instance:

```
class ArticleAdmin(admin.ModelAdmin):
    def save_formset(self, request, form, formset, change):
        instances = formset.save(commit=False)
        for instance in instances:
            instance.user = request.user
            instance.save()
        formset.save_m2m()
```

ModelAdmin.**get_ordering**(*self*, *request*)
New in version 1.4: *Please see the release notes* The `get_ordering` method takes a‘‘request‘‘ as parameter and is expected to return a `list` or `tuple` for ordering similar to the `ordering` attribute. For example:

```
class PersonAdmin(ModelAdmin):

    def get_ordering(self, request):
        if request.user.is_superuser:
            return ['name', 'rank']
        else:
            return ['name']
```

ModelAdmin.**save_related**(*self*, *request*, *form*, *formsets*, *change*)
New in version 1.4: *Please see the release notes* The `save_related` method is given the `HttpRequest`, the parent `ModelForm` instance, the list of inline formsets and a boolean value based on whether the parent is

being added or changed. Here you can do any pre- or post-save operations for objects related to the parent. Note that at this point the parent object and its form have already been saved.

ModelAdmin.**get_readonly_fields**(*self*, *request*, *obj=None*)
The get_readonly_fields method is given the HttpRequest and the obj being edited (or None on an add form) and is expected to return a list or tuple of field names that will be displayed as read-only, as described above in the ModelAdmin.readonly_fields section.

ModelAdmin.**get_prepopulated_fields**(*self*, *request*, *obj=None*)
New in version 1.4: *Please see the release notes* The get_prepopulated_fields method is given the HttpRequest and the obj being edited (or None on an add form) and is expected to return a dictionary, as described above in the ModelAdmin.prepopulated_fields section.

ModelAdmin.**get_list_display**(*self*, *request*)
New in version 1.4: *Please see the release notes* The get_list_display method is given the HttpRequest and is expected to return a list or tuple of field names that will be displayed on the changelist view as described above in the ModelAdmin.list_display section.

ModelAdmin.**get_list_display_links**(*self*, *request*, *list_display*)
New in version 1.4: *Please see the release notes* The get_list_display_links method is given the HttpRequest and the list or tuple returned by ModelAdmin.get_list_display(). It is expected to return a list or tuple of field names on the changelist that will be linked to the change view, as described in the ModelAdmin.list_display_links section.

ModelAdmin.**get_urls**(*self*)
The get_urls method on a ModelAdmin returns the URLs to be used for that ModelAdmin in the same way as a URLconf. Therefore you can extend them as documented in *URL dispatcher*:

```
class MyModelAdmin(admin.ModelAdmin):
    def get_urls(self):
        urls = super(MyModelAdmin, self).get_urls()
        my_urls = patterns('',
            (r'^my_view/$', self.my_view)
        )
        return my_urls + urls

    def my_view(self, request):
        # custom view which should return an HttpResponse
        pass
```

Note: Notice that the custom patterns are included *before* the regular admin URLs: the admin URL patterns are very permissive and will match nearly anything, so you'll usually want to prepend your custom URLs to the built-in ones.

In this example, my_view will be accessed at /admin/myapp/mymodel/my_view/ (assuming the admin URLs are included at /admin/.)

However, the self.my_view function registered above suffers from two problems:

- It will *not* perform any permission checks, so it will be accessible to the general public.
- It will *not* provide any header details to prevent caching. This means if the page retrieves data from the database, and caching middleware is active, the page could show outdated information.

Since this is usually not what you want, Django provides a convenience wrapper to check permissions and mark the view as non-cacheable. This wrapper is AdminSite.admin_view() (i.e. self.admin_site.admin_view inside a ModelAdmin instance); use it like so:

```
class MyModelAdmin(admin.ModelAdmin):
    def get_urls(self):
        urls = super(MyModelAdmin, self).get_urls()
        my_urls = patterns('',
            (r'^my_view/$', self.admin_site.admin_view(self.my_view))
        )
        return my_urls + urls
```

Notice the wrapped view in the fifth line above:

```
(r'^my_view/$', self.admin_site.admin_view(self.my_view))
```

This wrapping will protect `self.my_view` from unauthorized access and will apply the `django.views.decorators.cache.never_cache` decorator to make sure it is not cached if the cache middleware is active.

If the page is cacheable, but you still want the permission check to be performed, you can pass a `cacheable=True` argument to `AdminSite.admin_view()`:

```
(r'^my_view/$', self.admin_site.admin_view(self.my_view, cacheable=True))
```

`ModelAdmin.`**`formfield_for_foreignkey`**(*self*, *db_field*, *request*, ***kwargs*)

The `formfield_for_foreignkey` method on a `ModelAdmin` allows you to override the default formfield for a foreign keys field. For example, to return a subset of objects for this foreign key field based on the user:

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        if db_field.name == "car":
            kwargs["queryset"] = Car.objects.filter(owner=request.user)
        return super(MyModelAdmin, self).formfield_for_foreignkey(db_field, request, **kwargs)
```

This uses the `HttpRequest` instance to filter the `Car` foreign key field to only display the cars owned by the `User` instance.

`ModelAdmin.`**`formfield_for_manytomany`**(*self*, *db_field*, *request*, ***kwargs*)

Like the `formfield_for_foreignkey` method, the `formfield_for_manytomany` method can be overridden to change the default formfield for a many to many field. For example, if an owner can own multiple cars and cars can belong to multiple owners – a many to many relationship – you could filter the `Car` foreign key field to only display the cars owned by the `User`:

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_manytomany(self, db_field, request, **kwargs):
        if db_field.name == "cars":
            kwargs["queryset"] = Car.objects.filter(owner=request.user)
        return super(MyModelAdmin, self).formfield_for_manytomany(db_field, request, **kwargs)
```

`ModelAdmin.`**`formfield_for_choice_field`**(*self*, *db_field*, *request*, ***kwargs*)

Like the `formfield_for_foreignkey` and `formfield_for_manytomany` methods, the `formfield_for_choice_field` method can be overridden to change the default formfield for a field that has declared choices. For example, if the choices available to a superuser should be different than those available to regular staff, you could proceed as follows:

```
class MyModelAdmin(admin.ModelAdmin):
    def formfield_for_choice_field(self, db_field, request, **kwargs):
        if db_field.name == "status":
            kwargs['choices'] = (
                ('accepted', 'Accepted'),
                ('denied', 'Denied'),
```

```
            )
            if request.user.is_superuser:
                kwargs['choices'] += (('ready', 'Ready for deployment'),)
        return super(MyModelAdmin, self).formfield_for_choice_field(db_field, request, **kwargs)
```

ModelAdmin.**get_changelist**(*self*, *request*, ***kwargs*)
Returns the Changelist class to be used for listing. By default, `django.contrib.admin.views.main.ChangeList` is used. By inheriting this class you can change the behavior of the listing.

ModelAdmin.**has_add_permission**(*self*, *request*)
Should return `True` if adding an object is permitted, `False` otherwise.

ModelAdmin.**has_change_permission**(*self*, *request*, *obj=None*)
Should return `True` if editing obj is permitted, `False` otherwise. If obj is `None`, should return `True` or `False` to indicate whether editing of objects of this type is permitted in general (e.g., `False` will be interpreted as meaning that the current user is not permitted to edit any object of this type).

ModelAdmin.**has_delete_permission**(*self*, *request*, *obj=None*)
Should return `True` if deleting obj is permitted, `False` otherwise. If obj is `None`, should return `True` or `False` to indicate whether deleting objects of this type is permitted in general (e.g., `False` will be interpreted as meaning that the current user is not permitted to delete any object of this type).

ModelAdmin.**queryset**(*self*, *request*)
The `queryset` method on a `ModelAdmin` returns a `QuerySet` of all model instances that can be edited by the admin site. One use case for overriding this method is to show objects owned by the logged-in user:

```
class MyModelAdmin(admin.ModelAdmin):
    def queryset(self, request):
        qs = super(MyModelAdmin, self).queryset(request)
        if request.user.is_superuser:
            return qs
        return qs.filter(author=request.user)
```

ModelAdmin.**message_user**(*request*, *message*)
Sends a message to the user. The default implementation creates a message using the `django.contrib.messages` backend. See the *custom ModelAdmin example*.

ModelAdmin.**get_paginator**(*queryset*, *per_page*, *orphans=0*, *allow_empty_first_page=True*)
New in version 1.3: *Please see the release notes* Returns an instance of the paginator to use for this view. By default, instantiates an instance of `paginator`.

Other methods

ModelAdmin.**add_view**(*self*, *request*, *form_url=''*, *extra_context=None*)
Django view for the model instance addition page. See note below.

ModelAdmin.**change_view**(*self*, *request*, *object_id*, *form_url=''*, *extra_context=None*)
Django view for the model instance edition page. See note below. Changed in version 1.4: *Please see the release notes* The `form_url` parameter was added.

ModelAdmin.**changelist_view**(*self*, *request*, *extra_context=None*)
Django view for the model instances change list/actions page. See note below.

ModelAdmin.**delete_view**(*self*, *request*, *object_id*, *extra_context=None*)
Django view for the model instance(s) deletion confirmation page. See note below.

ModelAdmin.**history_view**(*self*, *request*, *object_id*, *extra_context=None*)
Django view for the page that shows the modification history for a given model instance.

Unlike the hook-type `ModelAdmin` methods detailed in the previous section, these five methods are in reality designed to be invoked as Django views from the admin application URL dispatching handler to render the pages that deal with model instances CRUD operations. As a result, completely overriding these methods will significantly change the behavior of the admin application.

One common reason for overriding these methods is to augment the context data that is provided to the template that renders the view. In the following example, the change view is overridden so that the rendered template is provided some extra mapping data that would not otherwise be available:

```
class MyModelAdmin(admin.ModelAdmin):

    # A template for a very customized change view:
    change_form_template = 'admin/myapp/extras/openstreetmap_change_form.html'

    def get_osm_info(self):
        # ...
        pass

    def change_view(self, request, object_id, extra_context=None):
        extra_context = extra_context or {}
        extra_context['osm_data'] = self.get_osm_info()
        return super(MyModelAdmin, self).change_view(request, object_id,
            extra_context=extra_context)
```

New in version 1.4: *Please see the release notes* These views now return `TemplateResponse` instances which allow you to easily customize the response data before rendering. For more details, see the *TemplateResponse documentation*.

`ModelAdmin` media definitions

There are times where you would like add a bit of CSS and/or JavaScript to the add/change views. This can be accomplished by using a Media inner class on your `ModelAdmin`:

```
class ArticleAdmin(admin.ModelAdmin):
    class Media:
        css = {
            "all": ("my_styles.css",)
        }
        js = ("my_code.js",)
```

Changed in version 1.3: *Please see the release notes* The *staticfiles app* prepends `STATIC_URL` (or `MEDIA_URL` if `STATIC_URL` is `None`) to any media paths. The same rules apply as *regular media definitions on forms*.

Django admin Javascript makes use of the jQuery library. To avoid conflicts with user-supplied scripts or libraries, Django's jQuery is namespaced as `django.jQuery`. If you want to use jQuery in your own admin JavaScript without including a second copy, you can use the `django.jQuery` object on changelist and add/edit views.

If you require the jQuery library to be in the global namespace, for example when using third-party jQuery plugins, or need a newer version of jQuery, you will have to include your own copy of jQuery.

Adding custom validation to the admin

Adding custom validation of data in the admin is quite easy. The automatic admin interface reuses `django.forms`, and the `ModelAdmin` class gives you the ability define your own form:

```
class ArticleAdmin(admin.ModelAdmin):
    form = MyArticleAdminForm
```

`MyArticleAdminForm` can be defined anywhere as long as you import where needed. Now within your form you can add your own custom validation for any field:

```
class MyArticleAdminForm(forms.ModelForm):
    class Meta:
        model = Article

    def clean_name(self):
        # do something that validates your data
        return self.cleaned_data["name"]
```

It is important you use a `ModelForm` here otherwise things can break. See the *forms* documentation on *custom validation* and, more specifically, the *model form validation notes* for more information.

`InlineModelAdmin` objects

class `InlineModelAdmin`

class `TabularInline`

class `StackedInline`

The admin interface has the ability to edit models on the same page as a parent model. These are called inlines. Suppose you have these two models:

```
class Author(models.Model):
   name = models.CharField(max_length=100)

class Book(models.Model):
   author = models.ForeignKey(Author)
   title = models.CharField(max_length=100)
```

You can edit the books authored by an author on the author page. You add inlines to a model by specifying them in a `ModelAdmin.inlines`:

```
class BookInline(admin.TabularInline):
    model = Book

class AuthorAdmin(admin.ModelAdmin):
    inlines = [
        BookInline,
    ]
```

Django provides two subclasses of `InlineModelAdmin` and they are:

- `TabularInline`
- `StackedInline`

The difference between these two is merely the template used to render them.

`InlineModelAdmin` options

`InlineModelAdmin` shares many of the same features as `ModelAdmin`, and adds some of its own (the shared features are actually defined in the `BaseModelAdmin` superclass). The shared features are:

- `form`

- fieldsets
- fields
- formfield_overrides
- exclude
- filter_horizontal
- filter_vertical
- prepopulated_fields
- radio_fields
- readonly_fields
- raw_id_fields
- formfield_for_foreignkey()
- formfield_for_manytomany()

New in version 1.3: *Please see the release notes*

- ordering
- queryset()

New in version 1.4: *Please see the release notes*

- has_add_permission()
- has_change_permission()
- has_delete_permission()

The InlineModelAdmin class adds:

InlineModelAdmin.**model**
The model which the inline is using. This is required.

InlineModelAdmin.**fk_name**
The name of the foreign key on the model. In most cases this will be dealt with automatically, but fk_name must be specified explicitly if there are more than one foreign key to the same parent model.

InlineModelAdmin.**formset**
This defaults to BaseInlineFormSet. Using your own formset can give you many possibilities of customization. Inlines are built around *model formsets*.

InlineModelAdmin.**form**
The value for form defaults to ModelForm. This is what is passed through to inlineformset_factory when creating the formset for this inline.

InlineModelAdmin.**extra**
This controls the number of extra forms the formset will display in addition to the initial forms. See the *formsets documentation* for more information.

For users with JavaScript-enabled browsers, an "Add another" link is provided to enable any number of additional inlines to be added in addition to those provided as a result of the extra argument.

The dynamic link will not appear if the number of currently displayed forms exceeds max_num, or if the user does not have JavaScript enabled.

InlineModelAdmin.**max_num**
: This controls the maximum number of forms to show in the inline. This doesn't directly correlate to the number of objects, but can if the value is small enough. See *Limiting the number of editable objects* for more information.

InlineModelAdmin.**raw_id_fields**
: By default, Django's admin uses a select-box interface (<select>) for fields that are `ForeignKey`. Sometimes you don't want to incur the overhead of having to select all the related instances to display in the drop-down.

  `raw_id_fields` is a list of fields you would like to change into a `Input` widget for either a `ForeignKey` or `ManyToManyField`:

```
class BookInline(admin.TabularInline):
    model = Book
    raw_id_fields = ("pages",)
```

InlineModelAdmin.**template**
: The template used to render the inline on the page.

InlineModelAdmin.**verbose_name**
: An override to the `verbose_name` found in the model's inner `Meta` class.

InlineModelAdmin.**verbose_name_plural**
: An override to the `verbose_name_plural` found in the model's inner `Meta` class.

InlineModelAdmin.**can_delete**
: Specifies whether or not inline objects can be deleted in the inline. Defaults to `True`.

Working with a model with two or more foreign keys to the same parent model

It is sometimes possible to have more than one foreign key to the same model. Take this model for instance:

```
class Friendship(models.Model):
    to_person = models.ForeignKey(Person, related_name="friends")
    from_person = models.ForeignKey(Person, related_name="from_friends")
```

If you wanted to display an inline on the `Person` admin add/change pages you need to explicitly define the foreign key since it is unable to do so automatically:

```
class FriendshipInline(admin.TabularInline):
    model = Friendship
    fk_name = "to_person"

class PersonAdmin(admin.ModelAdmin):
    inlines = [
        FriendshipInline,
    ]
```

Working with many-to-many models

By default, admin widgets for many-to-many relations will be displayed on whichever model contains the actual reference to the `ManyToManyField`. Depending on your `ModelAdmin` definition, each many-to-many field in your model will be represented by a standard HTML `<select multiple>`, a horizontal or vertical filter, or a `raw_id_admin` widget. However, it is also possible to replace these widgets with inlines.

Suppose we have the following models:

```
class Person(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, related_name='groups')
```

If you want to display many-to-many relations using an inline, you can do so by defining an `InlineModelAdmin` object for the relationship:

```
class MembershipInline(admin.TabularInline):
    model = Group.members.through

class PersonAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]

class GroupAdmin(admin.ModelAdmin):
    inlines = [
        MembershipInline,
    ]
    exclude = ('members',)
```

There are two features worth noting in this example.

Firstly - the `MembershipInline` class references `Group.members.through`. The `through` attribute is a reference to the model that manages the many-to-many relation. This model is automatically created by Django when you define a many-to-many field.

Secondly, the `GroupAdmin` must manually exclude the `members` field. Django displays an admin widget for a many-to-many field on the model that defines the relation (in this case, `Group`). If you want to use an inline model to represent the many-to-many relationship, you must tell Django's admin to *not* display this widget - otherwise you will end up with two widgets on your admin page for managing the relation.

In all other respects, the `InlineModelAdmin` is exactly the same as any other. You can customize the appearance using any of the normal `ModelAdmin` properties.

Working with many-to-many intermediary models

When you specify an intermediary model using the `through` argument to a `ManyToManyField`, the admin will not display a widget by default. This is because each instance of that intermediary model requires more information than could be displayed in a single widget, and the layout required for multiple widgets will vary depending on the intermediate model.

However, we still want to be able to edit that information inline. Fortunately, this is easy to do with inline admin models. Suppose we have the following models:

```
class Person(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
```

```
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

The first step in displaying this intermediate model in the admin is to define an inline class for the `Membership` model:

```
class MembershipInline(admin.TabularInline):
    model = Membership
    extra = 1
```

This simple example uses the default `InlineModelAdmin` values for the `Membership` model, and limits the extra add forms to one. This could be customized using any of the options available to `InlineModelAdmin` classes.

Now create admin views for the `Person` and `Group` models:

```
class PersonAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)

class GroupAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)
```

Finally, register your `Person` and `Group` models with the admin site:

```
admin.site.register(Person, PersonAdmin)
admin.site.register(Group, GroupAdmin)
```

Now your admin site is set up to edit `Membership` objects inline from either the `Person` or the `Group` detail pages.

Using generic relations as an inline

It is possible to use an inline with generically related objects. Let's say you have the following models:

```
class Image(models.Model):
    image = models.ImageField(upload_to="images")
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey("content_type", "object_id")

class Product(models.Model):
    name = models.CharField(max_length=100)
```

If you want to allow editing and creating `Image` instance on the `Product` add/change views you can use `GenericTabularInline` or `GenericStackedInline` (both subclasses of `GenericInlineModelAdmin`) provided by `django.contrib.contenttypes.generic`, they implement tabular and stacked visual layouts for the forms representing the inline objects respectively just like their non-generic counterparts and behave just like any other inline. In your `admin.py` for this example app:

```
from django.contrib import admin
from django.contrib.contenttypes import generic

from myproject.myapp.models import Image, Product

class ImageInline(generic.GenericTabularInline):
    model = Image

class ProductAdmin(admin.ModelAdmin):
    inlines = [
        ImageInline,
```

```
    ]

admin.site.register(Product, ProductAdmin)
```

See the *contenttypes documentation* for more specific information.

Overriding admin templates

It is relatively easy to override many of the templates which the admin module uses to generate the various pages of an admin site. You can even override a few of these templates for a specific app, or a specific model.

Set up your projects admin template directories

The admin template files are located in the `contrib/admin/templates/admin` directory.

In order to override one or more of them, first create an `admin` directory in your project's `templates` directory. This can be any of the directories you specified in `TEMPLATE_DIRS`.

Within this `admin` directory, create sub-directories named after your app. Within these app subdirectories create sub-directories named after your models. Note, that the admin app will lowercase the model name when looking for the directory, so make sure you name the directory in all lowercase if you are going to run your app on a case-sensitive filesystem.

To override an admin template for a specific app, copy and edit the template from the `django/contrib/admin/templates/admin` directory, and save it to one of the directories you just created.

For example, if we wanted to add a tool to the change list view for all the models in an app named `my_app`, we would copy `contrib/admin/templates/admin/change_list.html` to the `templates/admin/my_app/` directory of our project, and make any necessary changes.

If we wanted to add a tool to the change list view for only a specific model named 'Page', we would copy that same file to the `templates/admin/my_app/page` directory of our project.

Overriding vs. replacing an admin template

Because of the modular design of the admin templates, it is usually neither necessary nor advisable to replace an entire template. It is almost always better to override only the section of the template which you need to change.

To continue the example above, we want to add a new link next to the `History` tool for the `Page` model. After looking at `change_form.html` we determine that we only need to override the `object-tools` block. Therefore here is our new `change_form.html` :

```
{% extends "admin/change_form.html" %}
{% load i18n %}
{% block object-tools %}
{% if change %}{% if not is_popup %}
  <ul class="object-tools">
    <li><a href="history/" class="historylink">{% trans "History" %}</a></li>
    <li><a href="mylink/" class="historylink">My Link</a></li>
    {% if has_absolute_url %}
        <li><a href="../../../r/{{ content_type_id }}/{{ object_id }}/" class="viewsitelink">
            {% trans "View on site" %}</a>
        </li>
    {% endif%}
  </ul>
```

```
{% endif %}{% endif %}
{% endblock %}
```

And that's it! If we placed this file in the `templates/admin/my_app` directory, our link would appear on every model's change form.

Templates which may be overridden per app or model

Not every template in `contrib/admin/templates/admin` may be overridden per app or per model. The following can:

- `app_index.html`
- `change_form.html`
- `change_list.html`
- `delete_confirmation.html`
- `object_history.html`

For those templates that cannot be overridden in this way, you may still override them for your entire project. Just place the new version in your `templates/admin` directory. This is particularly useful to create custom 404 and 500 pages.

Note: Some of the admin templates, such as `change_list_request.html` are used to render custom inclusion tags. These may be overridden, but in such cases you are probably better off creating your own version of the tag in question and giving it a different name. That way you can use it selectively.

Root and login templates

If you wish to change the index, login or logout templates, you are better off creating your own `AdminSite` instance (see below), and changing the `AdminSite.index_template` , `AdminSite.login_template` or `AdminSite.logout_template` properties.

`AdminSite` objects

class `AdminSite`(*name='admin'*)

A Django administrative site is represented by an instance of `django.contrib.admin.sites.AdminSite`; by default, an instance of this class is created as `django.contrib.admin.site` and you can register your models and `ModelAdmin` instances with it.

If you'd like to set up your own administrative site with custom behavior, however, you're free to subclass `AdminSite` and override or add anything you like. Then, simply create an instance of your `AdminSite` subclass (the same way you'd instantiate any other Python class), and register your models and `ModelAdmin` subclasses with it instead of using the default.

When constructing an instance of an `AdminSite`, you are able to provide a unique instance name using the `name` argument to the constructor. This instance name is used to identify the instance, especially when *reversing admin URLs*. If no instance name is provided, a default instance name of `admin` will be used.

`AdminSite` attributes

Templates can override or extend base admin templates as described in Overriding Admin Templates.

`AdminSite.`**`index_template`**
Path to a custom template that will be used by the admin site main index view.

`AdminSite.`**`login_template`**
Path to a custom template that will be used by the admin site login view.

`AdminSite.`**`login_form`**
New in version 1.3: *Please see the release notes* Subclass of `AuthenticationForm` that will be used by the admin site login view.

`AdminSite.`**`logout_template`**
Path to a custom template that will be used by the admin site logout view.

`AdminSite.`**`password_change_template`**
Path to a custom template that will be used by the admin site password change view.

`AdminSite.`**`password_change_done_template`**
Path to a custom template that will be used by the admin site password change done view.

Hooking `AdminSite` instances into your URLconf

The last step in setting up the Django admin is to hook your `AdminSite` instance into your URLconf. Do this by pointing a given URL at the `AdminSite.urls` method.

In this example, we register the default `AdminSite` instance `django.contrib.admin.site` at the URL `/admin/`

```
# urls.py
from django.conf.urls import patterns, url, include
from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/', include(admin.site.urls)),
)
```

Above we used `admin.autodiscover()` to automatically load the `INSTALLED_APPS` admin.py modules.

In this example, we register the `AdminSite` instance `myproject.admin.admin_site` at the URL `/myadmin/`

```
# urls.py
from django.conf.urls import patterns, url, include
from myproject.admin import admin_site

urlpatterns = patterns('',
    (r'^myadmin/', include(admin_site.urls)),
)
```

There is really no need to use autodiscover when using your own `AdminSite` instance since you will likely be importing all the per-app admin.py modules in your `myproject.admin` module.

Multiple admin sites in the same URLconf

It's easy to create multiple instances of the admin site on the same Django-powered Web site. Just create multiple instances of `AdminSite` and root each one at a different URL.

In this example, the URLs `/basic-admin/` and `/advanced-admin/` feature separate versions of the admin site – using the `AdminSite` instances `myproject.admin.basic_site` and `myproject.admin.advanced_site`, respectively:

```
# urls.py
from django.conf.urls import patterns, url, include
from myproject.admin import basic_site, advanced_site

urlpatterns = patterns('',
    (r'^basic-admin/', include(basic_site.urls)),
    (r'^advanced-admin/', include(advanced_site.urls)),
)
```

`AdminSite` instances take a single argument to their constructor, their name, which can be anything you like. This argument becomes the prefix to the URL names for the purposes of *reversing them*. This is only necessary if you are using more than one `AdminSite`.

Adding views to admin sites

Just like `ModelAdmin`, `AdminSite` provides a `get_urls()` method that can be overridden to define additional views for the site. To add a new view to your admin site, extend the base `get_urls()` method to include a pattern for your new view.

Note: Any view you render that uses the admin templates, or extends the base admin template, should provide the `current_app` argument to `RequestContext` or `Context` when rendering the template. It should be set to either `self.name` if your view is on an `AdminSite` or `self.admin_site.name` if your view is on a `ModelAdmin`.

Adding a password-reset feature

You can add a password-reset feature to the admin site by adding a few lines to your URLconf. Specifically, add these four patterns:

```
url(r'^admin/password_reset/$', 'django.contrib.auth.views.password_reset', name='admin_password_rese
(r'^admin/password_reset/done/$', 'django.contrib.auth.views.password_reset_done'),
(r'^reset/(?P<uidb36>[0-9A-Za-z]+)-(?P<token>.+)/$', 'django.contrib.auth.views.password_reset_confi
(r'^reset/done/$', 'django.contrib.auth.views.password_reset_complete'),
```

(This assumes you've added the admin at `admin/` and requires that you put the URLs starting with `^admin/` before the line that includes the admin app itself). Changed in version 1.4: *Please see the release notes* The presence of the `admin_password_reset` named URL will cause a "forgotten your password?" link to appear on the default admin log-in page under the password box.

Reversing admin URLs

When an `AdminSite` is deployed, the views provided by that site are accessible using Django's *URL reversing system*.

The `AdminSite` provides the following named URL patterns:

Page	URL name	Parameters
Index	index	
Logout	logout	
Password change	password_change	
Password change done	password_change_done	
i18n javascript	jsi18n	
Application index page	app_list	app_label
Redirect to object's page	view_on_site	content_type_id, object_id

Each ModelAdmin instance provides an additional set of named URLs:

Page	URL name	Parameters
Changelist	{{ app_label }}_{{ model_name }}_changelist	
Add	{{ app_label }}_{{ model_name }}_add	
History	{{ app_label }}_{{ model_name }}_history	object_id
Delete	{{ app_label }}_{{ model_name }}_delete	object_id
Change	{{ app_label }}_{{ model_name }}_change	object_id

These named URLs are registered with the application namespace admin, and with an instance namespace corresponding to the name of the Site instance.

So - if you wanted to get a reference to the Change view for a particular Choice object (from the polls application) in the default admin, you would call:

```
>>> from django.core import urlresolvers
>>> c = Choice.objects.get(...)
>>> change_url = urlresolvers.reverse('admin:polls_choice_change', args=(c.id,))
```

This will find the first registered instance of the admin application (whatever the instance name), and resolve to the view for changing poll.Choice instances in that instance.

If you want to find a URL in a specific admin instance, provide the name of that instance as a current_app hint to the reverse call. For example, if you specifically wanted the admin view from the admin instance named custom, you would need to call:

```
>>> change_url = urlresolvers.reverse('custom:polls_choice_change', args=(c.id,))
```

For more details, see the documentation on *reversing namespaced URLs*. New in version 1.4: *Please see the release notes* To allow easier reversing of the admin urls in templates, Django provides an admin_urlname filter which takes an action as argument:

```
{% load admin_urls %}
<a href="{% url opts|admin_urlname:'add' %}">Add user</a>
<a href="{% url opts|admin_urlname:'delete' user.pk %}">Delete this user</a>
```

The action in the examples above match the last part of the URL names for ModelAdmin instances described above. The opts variable can be any object which has an app_label and module_name and is usually supplied by the admin views for the current model.

6.4.2 django.contrib.auth

See *User authentication in Django*.

6.4.3 Django's comments framework

Django includes a simple, yet customizable comments framework. The built-in comments framework can be used to attach comments to any model, so you can use it for comments on blog entries, photos, book chapters, or anything

else.

Note: If you used to use Django's older (undocumented) comments framework, you'll need to upgrade. See the *upgrade guide* for instructions.

Quick start guide

To get started using the `comments` app, follow these steps:

1. Install the comments framework by adding `'django.contrib.comments'` to `INSTALLED_APPS`.
2. Run `manage.py syncdb` so that Django will create the comment tables.
3. Add the comment app's URLs to your project's `urls.py`:

```
urlpatterns = patterns('',
    ...
    (r'^comments/', include('django.contrib.comments.urls')),
    ...
)
```

4. Use the comment template tags below to embed comments in your templates.

You might also want to examine *Comment settings*.

Comment template tags

You'll primarily interact with the comment system through a series of template tags that let you embed comments and generate forms for your users to post them.

Like all custom template tag libraries, you'll need to *load the custom tags* before you can use them:

```
{% load comments %}
```

Once loaded you can use the template tags below.

Specifying which object comments are attached to

Django's comments are all "attached" to some parent object. This can be any instance of a Django model. Each of the tags below gives you a couple of different ways you can specify which object to attach to:

1. Refer to the object directly – the more common method. Most of the time, you'll have some object in the template's context you want to attach the comment to; you can simply use that object.

   For example, in a blog entry page that has a variable named `entry`, you could use the following to load the number of comments:

   ```
   {% get_comment_count for entry as comment_count %}.
   ```

2. Refer to the object by content-type and object id. You'd use this method if you, for some reason, don't actually have direct access to the object.

   Following the above example, if you knew the object ID was `14` but didn't have access to the actual object, you could do something like:

   ```
   {% get_comment_count for blog.entry 14 as comment_count %}
   ```

   In the above, `blog.entry` is the app label and (lower-cased) model name of the model class.

Displaying comments

To display a list of comments, you can use the template tags `render_comment_list` or `get_comment_list`.

Quickly rendering a comment list The easiest way to display a list of comments for some object is by using `render_comment_list`:

```
{% render_comment_list for [object] %}
```

For example:

```
{% render_comment_list for event %}
```

This will render comments using a template named `comments/list.html`, a default version of which is included with Django.

Rendering a custom comment list To get the list of comments for some object, use `get_comment_list`:

```
{% get_comment_list for [object] as [varname] %}
```

For example:

```
{% get_comment_list for event as comment_list %}
{% for comment in comment_list %}
    ...
{% endfor %}
```

This returns a list of `Comment` objects; see *the comment model documentation* for details.

Linking to comments

To provide a permalink to a specific comment, use `get_comment_permalink`:

```
{% get_comment_permalink comment_obj [format_string] %}
```

By default, the named anchor that will be appended to the URL will be the letter 'c' followed by the comment id, for example 'c82'. You may specify a custom format string if you wish to override this behavior:

```
{% get_comment_permalink comment "#c%(id)s-by-%(user_name)s"%}
```

The format string is a standard python format string. Valid mapping keys include any attributes of the comment object.

Regardless of whether you specify a custom anchor pattern, you must supply a matching named anchor at a suitable place in your template.

For example:

```
{% for comment in comment_list %}
    <a name="c{{ comment.id }}"></a>
    <a href="{% get_comment_permalink comment %}">
        permalink for comment #{{ forloop.counter }}
    </a>
    ...
{% endfor %}
```

Warning: There's a known bug in Safari/Webkit which causes the named anchor to be forgotten following a redirect. The practical impact for comments is that the Safari/webkit browsers will arrive at the correct page but will not scroll to the named anchor.

Counting comments

To count comments attached to an object, use `get_comment_count`:

```
{% get_comment_count for [object] as [varname]  %}
```

For example:

```
{% get_comment_count for event as comment_count %}

<p>This event has {{ comment_count }} comments.</p>
```

Displaying the comment post form

To show the form that users will use to post a comment, you can use `render_comment_form` or `get_comment_form`

Quickly rendering the comment form The easiest way to display a comment form is by using `render_comment_form`:

```
{% render_comment_form for [object] %}
```

For example:

```
{% render_comment_form for event %}
```

This will render comments using a template named `comments/form.html`, a default version of which is included with Django.

Rendering a custom comment form If you want more control over the look and feel of the comment form, you use use `get_comment_form` to get a *form object* that you can use in the template:

```
{% get_comment_form for [object] as [varname] %}
```

A complete form might look like:

```
{% get_comment_form for event as form %}
<table>
  <form action="{% comment_form_target %}" method="post">
    {% csrf_token %}
    {{ form }}
    <tr>
      <td colspan="2">
        <input type="submit" name="submit" value="Post">
        <input type="submit" name="preview" value="Preview">
      </td>
    </tr>
  </form>
</table>
```

Be sure to read the notes on the comment form, below, for some special considerations you'll need to make if you're using this approach.

Getting the comment form target You may have noticed that the above example uses another template tag – `comment_form_target` – to actually get the `action` attribute of the form. This will always return the correct URL that comments should be posted to; you'll always want to use it like above:

```
<form action="{% comment_form_target %}" method="post">
```

Redirecting after the comment post To specify the URL you want to redirect to after the comment has been posted, you can include a hidden form input called `next` in your comment form. For example:

```
<input type="hidden" name="next" value="{% url 'my_comment_was_posted' %}" />
```

Notes on the comment form

The form used by the comment system has a few important anti-spam attributes you should know about:

- It contains a number of hidden fields that contain timestamps, information about the object the comment should be attached to, and a "security hash" used to validate this information. If someone tampers with this data – something comment spammers will try – the comment submission will fail.

  If you're rendering a custom comment form, you'll need to make sure to pass these values through unchanged.

- The timestamp is used to ensure that "reply attacks" can't continue very long. Users who wait too long between requesting the form and posting a comment will have their submissions refused.

- The comment form includes a "honeypot" field. It's a trap: if any data is entered in that field, the comment will be considered spam (spammers often automatically fill in all fields in an attempt to make valid submissions).

  The default form hides this field with a piece of CSS and further labels it with a warning field; if you use the comment form with a custom template you should be sure to do the same.

The comments app also depends on the more general *Cross Site Request Forgery protection* that comes with Django. As described in the documentation, it is best to use `CsrfViewMiddleware`. However, if you are not using that, you will need to use the `csrf_protect` decorator on any views that include the comment form, in order for those views to be able to output the CSRF token and cookie.

More information

The built-in comment models

class `Comment`

Django's built-in comment model. Has the following fields:

`content_object`

A `GenericForeignKey` attribute pointing to the object the comment is attached to. You can use this to get at the related object (i.e. `my_comment.content_object`).

Since this field is a `GenericForeignKey`, it's actually syntactic sugar on top of two underlying attributes, described below.

`content_type`

A `ForeignKey` to `ContentType`; this is the type of the object the comment is attached to.

object_pk
A `TextField` containing the primary key of the object the comment is attached to.

site
A `ForeignKey` to the `Site` on which the comment was posted.

user
A `ForeignKey` to the `User` who posted the comment. May be blank if the comment was posted by an unauthenticated user.

user_name
The name of the user who posted the comment.

user_email
The email of the user who posted the comment.

user_url
The URL entered by the person who posted the comment.

comment
The actual content of the comment itself.

submit_date
The date the comment was submitted.

ip_address
The IP address of the user posting the comment.

is_public
`False` if the comment is in moderation (see *Generic comment moderation*); If `True`, the comment will be displayed on the site.

is_removed
`True` if the comment was removed. Used to keep track of removed comments instead of just deleting them.

Comment settings

These settings configure the behavior of the comments framework:

COMMENTS_HIDE_REMOVED If `True` (default), removed comments will be excluded from comment lists/counts (as taken from template tags). Otherwise, the template author is responsible for some sort of a "this comment has been removed by the site staff" message.

COMMENT_MAX_LENGTH The maximum length of the comment field, in characters. Comments longer than this will be rejected. Defaults to 3000.

COMMENTS_APP An app which provides *customization of the comments framework*. Use the same dotted-string notation as in `INSTALLED_APPS`. Your custom `COMMENTS_APP` must also be listed in `INSTALLED_APPS`.

Signals sent by the comments app

The comment app sends a series of *signals* to allow for comment moderation and similar activities. See *the introduction to signals* for information about how to register for and receive these signals.

comment_will_be_posted
`django.contrib.comments.signals.`**`comment_will_be_posted`**
Sent just before a comment will be saved, after it's been sanity checked and submitted. This can be used to modify the comment (in place) with posting details or other such actions.

If any receiver returns `False` the comment will be discarded and a 403 (not allowed) response will be returned.

This signal is sent at more or less the same time (just before, actually) as the `Comment` object's `pre_save` signal.

Arguments sent with this signal:

`sender` The comment model.

`comment` The comment instance about to be posted. Note that it won't have been saved into the database yet, so it won't have a primary key, and any relations might not work correctly yet.

`request` The `HttpRequest` that posted the comment.

comment_was_posted
`django.contrib.comments.signals.`**`comment_was_posted`**
Sent just after the comment is saved.

Arguments sent with this signal:

`sender` The comment model.

`comment` The comment instance that was posted. Note that it will have already been saved, so if you modify it you'll need to call `save()` again.

`request` The `HttpRequest` that posted the comment.

comment_was_flagged
`django.contrib.comments.signals.`**`comment_was_flagged`**
Sent after a comment was "flagged" in some way. Check the flag to see if this was a user requesting removal of a comment, a moderator approving/removing a comment, or some other custom user flag.

Arguments sent with this signal:

`sender` The comment model.

`comment` The comment instance that was posted. Note that it will have already been saved, so if you modify it you'll need to call `save()` again.

`flag` The `CommentFlag` that's been attached to the comment.

`created` `True` if this is a new flag; `False` if it's a duplicate flag.

`request` The `HttpRequest` that posted the comment.

Upgrading from Django's previous comment system

Prior versions of Django included an outdated, undocumented comment system. Users who reverse-engineered this framework will need to upgrade to use the new comment system; this guide explains how.

The main changes from the old system are:

- This new system is documented.
- It uses modern Django features like *forms* and *modelforms*.
- It has a single `Comment` model instead of separate `FreeComment` and `Comment` models.

- Comments have "email" and "URL" fields.
- No ratings, photos and karma. This should only effect World Online.
- The `{% comment_form %}` tag no longer exists. Instead, there's now two functions: `{% get_comment_form %}`, which returns a form for posting a new comment, and `{% render_comment_form %}`, which renders said form using the `comments/form.html` template.
- The way comments are include in your URLconf have changed; you'll need to replace:

  ```
  (r'^comments/', include('django.contrib.comments.urls.comments')),
  ```

  with:

  ```
  (r'^comments/', include('django.contrib.comments.urls')),
  ```

Upgrading data The data models for Django's comment system have changed, as have the table names. Before you transfer your existing data into the new comments system, make sure that you have installed the new comments system as explained in the *quick start guide*. This will ensure that the new tables have been properly created.

To transfer your data into the new comments system, you'll need to directly run the following SQL:

```
BEGIN;

INSERT INTO django_comments
    (content_type_id, object_pk, site_id, user_name, user_email, user_url,
    comment, submit_date, ip_address, is_public, is_removed)
SELECT
    content_type_id, object_id, site_id, person_name, '', '', comment,
    submit_date, ip_address, is_public, not approved
FROM comments_freecomment;

INSERT INTO django_comments
    (content_type_id, object_pk, site_id, user_id, user_name, user_email,
    user_url, comment, submit_date, ip_address, is_public, is_removed)
SELECT
    content_type_id, object_id, site_id, user_id, '', '', '', comment,
    submit_date, ip_address, is_public, is_removed
FROM comments_comment;

UPDATE django_comments SET user_name = (
    SELECT username FROM auth_user
    WHERE django_comments.user_id = auth_user.id
) WHERE django_comments.user_id is not NULL;
UPDATE django_comments SET user_email = (
    SELECT email FROM auth_user
    WHERE django_comments.user_id = auth_user.id
) WHERE django_comments.user_id is not NULL;

COMMIT;
```

Customizing the comments framework

If the built-in comment framework doesn't quite fit your needs, you can extend the comment app's behavior to add custom data and logic. The comments framework lets you extend the built-in comment model, the built-in comment form, and the various comment views.

The COMMENTS_APP setting is where this customization begins. Set COMMENTS_APP to the name of the app you'd like to use to provide custom behavior. You'll use the same syntax as you'd use for INSTALLED_APPS, and the app given must also be in the INSTALLED_APPS list.

For example, if you wanted to use an app named my_comment_app, your settings file would contain:

```
INSTALLED_APPS = [
    ...
    'my_comment_app',
    ...
]

COMMENTS_APP = 'my_comment_app'
```

The app named in COMMENTS_APP provides its custom behavior by defining some module-level functions in the app's __init__.py. The *complete list of these functions* can be found below, but first let's look at a quick example.

An example custom comments app One of the most common types of customization is modifying the set of fields provided on the built-in comment model. For example, some sites that allow comments want the commentator to provide a title for their comment; the built-in comment model has no field for that title.

To make this kind of customization, we'll need to do three things:

1. Create a custom comment Model that adds on the "title" field.
2. Create a custom comment Form that also adds this "title" field.
3. Inform Django of these objects by defining a few functions in a custom COMMENTS_APP.

So, carrying on the example above, we're dealing with a typical app structure in the my_custom_app directory:

```
my_custom_app/
    __init__.py
    models.py
    forms.py
```

In the models.py we'll define a CommentWithTitle model:

```
from django.db import models
from django.contrib.comments.models import Comment

class CommentWithTitle(Comment):
    title = models.CharField(max_length=300)
```

Most custom comment models will subclass the Comment model. However, if you want to substantially remove or change the fields available in the Comment model, but don't want to rewrite the templates, you could try subclassing from BaseCommentAbstractModel.

Next, we'll define a custom comment form in forms.py. This is a little more tricky: we have to both create a form and override CommentForm.get_comment_model() and CommentForm.get_comment_create_data() to return deal with our custom title field:

```
from django import forms
from django.contrib.comments.forms import CommentForm
from my_comment_app.models import CommentWithTitle

class CommentFormWithTitle(CommentForm):
    title = forms.CharField(max_length=300)

    def get_comment_model(self):
```

```
        # Use our custom comment model instead of the built-in one.
        return CommentWithTitle

    def get_comment_create_data(self):
        # Use the data of the superclass, and add in the title field
        data = super(CommentFormWithTitle, self).get_comment_create_data()
        data['title'] = self.cleaned_data['title']
        return data
```

Django provides a couple of "helper" classes to make writing certain types of custom comment forms easier; see `django.contrib.comments.forms` for more.

Finally, we'll define a couple of methods in `my_custom_app/__init__.py` to point Django at these classes we've created:

```
from my_comments_app.models import CommentWithTitle
from my_comments_app.forms import CommentFormWithTitle

def get_model():
    return CommentWithTitle

def get_form():
    return CommentFormWithTitle
```

> **Warning:** Be careful not to create cyclic imports in your custom comments app. If you feel your comment configuration isn't being used as defined – for example, if your comment moderation policy isn't being applied – you may have a cyclic import problem.
> If you are having unexplained problems with comments behavior, check if your custom comments application imports (even indirectly) any module that itself imports Django's comments module.

The above process should take care of most common situations. For more advanced usage, there are additional methods you can define. Those are explained in the next section.

Custom comment app API The `django.contrib.comments` app defines the following methods; any custom comment app must define at least one of them. All are optional, however.

`get_model``()`
: Return the `Model` class to use for comments. This model should inherit from `django.contrib.comments.models.BaseCommentAbstractModel`, which defines necessary core fields.

  The default implementation returns `django.contrib.comments.models.Comment`.

`get_form``()`
: Return the `Form` class you want to use for creating, validating, and saving your comment model. Your custom comment form should accept an additional first argument, `target_object`, which is the object the comment will be attached to.

  The default implementation returns `django.contrib.comments.forms.CommentForm`.

  Note: The default comment form also includes a number of unobtrusive spam-prevention features (see *Notes on the comment form*). If replacing it with your own form, you may want to look at the source code for the built-in form and consider incorporating similar features.

`get_form_target``()`
: Return the URL for POSTing comments. This will be the `<form action>` attribute when rendering your

comment form.

The default implementation returns a reverse-resolved URL pointing to the `post_comment()` view.

Note: If you provide a custom comment model and/or form, but you want to use the default `post_comment()` view, you will need to be aware that it requires the model and form to have certain additional attributes and methods: see the `post_comment()` view documentation for details.

`get_flag_url``()`
Return the URL for the "flag this comment" view.

The default implementation returns a reverse-resolved URL pointing to the `django.contrib.comments.views.moderation.flag()` view.

`get_delete_url``()`
Return the URL for the "delete this comment" view.

The default implementation returns a reverse-resolved URL pointing to the `django.contrib.comments.views.moderation.delete()` view.

`get_approve_url``()`
Return the URL for the "approve this comment from moderation" view.

The default implementation returns a reverse-resolved URL pointing to the `django.contrib.comments.views.moderation.approve()` view.

Comment form classes

The `django.contrib.comments.forms` module contains a handful of forms you'll use when writing custom views dealing with comments, or when writing *custom comment apps*.

class `CommentForm`
The main comment form representing the standard, built-in way of handling submitted comments. This is the class used by all the views `django.contrib.comments` to handle submitted comments.

If you want to build custom views that are similar to Django's built-in comment handling views, you'll probably want to use this form.

Abstract comment forms for custom comment apps If you're building a *custom comment app*, you might want to replace *some* of the form logic but still rely on parts of the existing form.

`CommentForm` is actually composed of a couple of abstract base class forms that you can subclass to reuse pieces of the form handling logic:

class `CommentSecurityForm`
Handles the anti-spoofing protection aspects of the comment form handling.

This class contains the `content_type` and `object_pk` fields pointing to the object the comment is attached to, along with a `timestamp` and a `security_hash` of all the form data. Together, the timestamp and the security hash ensure that spammers can't "replay" form submissions and flood you with comments.

class `CommentDetailsForm`
Handles the details of the comment itself.

This class contains the `name`, `email`, `url`, and the `comment` field itself, along with the associated validation logic.

Generic comment moderation

Django's bundled comments application is extremely useful on its own, but the amount of comment spam circulating on the Web today essentially makes it necessary to have some sort of automatic moderation system in place for any application which makes use of comments. To make this easier to handle in a consistent fashion, `django.contrib.comments.moderation` provides a generic, extensible comment-moderation system which can be applied to any model or set of models which want to make use of Django's comment system.

Overview The entire system is contained within `django.contrib.comments.moderation`, and uses a two-step process to enable moderation for any given model:

1. A subclass of `CommentModerator` is defined which specifies the moderation options the model wants to enable.
2. The model is registered with the moderation system, passing in the model class and the class which specifies its moderation options.

A simple example is the best illustration of this. Suppose we have the following model, which would represent entries in a Weblog:

```
from django.db import models

class Entry(models.Model):
    title = models.CharField(maxlength=250)
    body = models.TextField()
    pub_date = models.DateTimeField()
    enable_comments = models.BooleanField()
```

Now, suppose that we want the following steps to be applied whenever a new comment is posted on an `Entry`:

1. If the `Entry`'s `enable_comments` field is `False`, the comment will simply be disallowed (i.e., immediately deleted).
2. If the `enable_comments` field is `True`, the comment will be allowed to save.
3. Once the comment is saved, an email should be sent to site staff notifying them of the new comment.

Accomplishing this is fairly straightforward and requires very little code:

```
from django.contrib.comments.moderation import CommentModerator, moderator

class EntryModerator(CommentModerator):
    email_notification = True
    enable_field = 'enable_comments'

moderator.register(Entry, EntryModerator)
```

The `CommentModerator` class pre-defines a number of useful moderation options which subclasses can enable or disable as desired, and `moderator` knows how to work with them to determine whether to allow a comment, whether to moderate a comment which will be allowed to post, and whether to email notifications of new comments.

Built-in moderation options

class `CommentModerator`

Most common comment-moderation needs can be handled by subclassing `CommentModerator` and changing the values of pre-defined attributes; the full range of built-in options is as follows.

`auto_close_field`

If this is set to the name of a `DateField` or `DateTimeField` on the model for which comments are

being moderated, new comments for objects of that model will be disallowed (immediately deleted) when a certain number of days have passed after the date specified in that field. Must be used in conjunction with `close_after`, which specifies the number of days past which comments should be disallowed. Default value is `None`.

`auto_moderate_field`
Like `auto_close_field`, but instead of outright deleting new comments when the requisite number of days have elapsed, it will simply set the `is_public` field of new comments to `False` before saving them. Must be used in conjunction with `moderate_after`, which specifies the number of days past which comments should be moderated. Default value is `None`.

`close_after`
If `auto_close_field` is used, this must specify the number of days past the value of the field specified by `auto_close_field` after which new comments for an object should be disallowed. Allowed values are `None`, 0 (which disallows comments immediately), or any positive integer. Default value is `None`.

`email_notification`
If `True`, any new comment on an object of this model which survives moderation (i.e., is not deleted) will generate an email to site staff. Default value is `False`.

`enable_field`
If this is set to the name of a `BooleanField` on the model for which comments are being moderated, new comments on objects of that model will be disallowed (immediately deleted) whenever the value of that field is `False` on the object the comment would be attached to. Default value is `None`.

`moderate_after`
If `auto_moderate_field` is used, this must specify the number of days past the value of the field specified by `auto_moderate_field` after which new comments for an object should be marked non-public. Allowed values are `None`, 0 (which moderates comments immediately), or any positive integer. Default value is `None`.

Simply subclassing `CommentModerator` and changing the values of these options will automatically enable the various moderation methods for any models registered using the subclass. Changed in version 1.3: *Please see the release notes* `moderate_after` and `close_after` now accept 0 as a valid value.

Adding custom moderation methods For situations where the built-in options listed above are not sufficient, subclasses of `CommentModerator` can also override the methods which actually perform the moderation, and apply any logic they desire. `CommentModerator` defines three methods which determine how moderation will take place; each method will be called by the moderation system and passed two arguments: `comment`, which is the new comment being posted, `content_object`, which is the object the comment will be attached to, and `request`, which is the `HttpRequest` in which the comment is being submitted:

`CommentModerator.`**`allow`**(*comment*, *content_object*, *request*)
Should return `True` if the comment should be allowed to post on the content object, and `False` otherwise (in which case the comment will be immediately deleted).

`CommentModerator.`**`email`**(*comment*, *content_object*, *request*)
If email notification of the new comment should be sent to site staff or moderators, this method is responsible for sending the email.

`CommentModerator.`**`moderate`**(*comment*, *content_object*, *request*)
Should return `True` if the comment should be moderated (in which case its `is_public` field will be set to `False` before saving), and `False` otherwise (in which case the `is_public` field will not be changed).

Registering models for moderation The moderation system, represented by `django.contrib.comments.moderation.moderator` is an instance of the class `Moderator`, which allows registration and "unregistration" of models via two methods:

moderator.**register**(*model_or_iterable*, *moderation_class*)
Takes two arguments: the first should be either a model class or list of model classes, and the second should be a subclass of CommentModerator, and register the model or models to be moderated using the options defined in the CommentModerator subclass. If any of the models are already registered for moderation, the exception AlreadyModerated will be raised.

moderator.**unregister**(*model_or_iterable*)
Takes one argument: a model class or list of model classes, and removes the model or models from the set of models which are being moderated. If any of the models are not currently being moderated, the exception NotModerated will be raised.

Customizing the moderation system Most use cases will work easily with simple subclassing of CommentModerator and registration with the provided Moderator instance, but customization of global moderation behavior can be achieved by subclassing Moderator and instead registering models with an instance of the subclass.

class Moderator
In addition to the Moderator.register() and Moderator.unregister() methods detailed above, the following methods on Moderator can be overridden to achieve customized behavior:

connect()
Determines how moderation is set up globally. The base implementation in Moderator does this by attaching listeners to the comment_will_be_posted and comment_was_posted signals from the comment models.

pre_save_moderation(*sender*, *comment*, *request*, ***kwargs*)
In the base implementation, applies all pre-save moderation steps (such as determining whether the comment needs to be deleted, or whether it needs to be marked as non-public or generate an email).

post_save_moderation(*sender*, *comment*, *request*, ***kwargs*)
In the base implementation, applies all post-save moderation steps (currently this consists entirely of deleting comments which were disallowed).

Example of using the built-in comments app

Follow the first three steps of the quick start guide in the *documentation*.

Now suppose, you have an app (blog) with a model (Post) to which you want to attach comments. Let's also suppose that you have a template called blog_detail.html where you want to display the comments list and comment form.

Template First, we should load the comment template tags in the blog_detail.html so that we can use its functionality. So just like all other custom template tag libraries:

```
{% load comments %}
```

Next, let's add the number of comments attached to the particular model instance of Post. For this we assume that a context variable object_pk is present which gives the id of the instance of Post.

The usage of the get_comment_count tag is like below:

```
{% get_comment_count for blog.post object_pk as comment_count %}
<p>{{ comment_count }} comments have been posted.</p>
```

If you have the instance (say entry) of the model (Post) available in the context, then you can refer to it directly:

```
{% get_comment_count for entry as comment_count %}
<p>{{ comment_count }} comments have been posted.</p>
```

Next, we can use the `render_comment_list` tag, to render all comments to the given instance (`entry`) by using the `comments/list.html` template:

```
{% render_comment_list for entry %}
```

Django will will look for the `list.html` under the following directories (for our example):

```
comments/blog/post/list.html
comments/blog/list.html
comments/list.html
```

To get a list of comments, we make use of the `get_comment_list` tag. Using this tag is very similar to the `get_comment_count` tag. We need to remember that `get_comment_list` returns a list of comments and hence we have to iterate through them to display them:

```
{% get_comment_list for blog.post object_pk as comment_list %}
{% for comment in comment_list %}
<p>Posted by: {{ comment.user_name }} on {{ comment.submit_date }}</p>
...
<p>Comment: {{ comment.comment }}</p>
...
{% endfor %}
```

Finally, we display the comment form, enabling users to enter their comments. There are two ways of doing so. The first is when you want to display the comments template available under your `comments/form.html`. The other method gives you a chance to customize the form.

The first method makes use of the `render_comment_form` tag. Its usage too is similar to the other three tags we have discussed above:

```
{% render_comment_form for entry %}
```

It looks for the `form.html` under the following directories (for our example):

```
comments/blog/post/form.html
comments/blog/form.html
comments/form.html
```

Since we customize the form in the second method, we make use of another tag called `comment_form_target`. This tag on rendering gives the URL where the comment form is posted. Without any *customization*, `comment_form_target` evaluates to `/comments/post/`. We use this tag in the form's `action` attribute.

The `get_comment_form` tag renders a `form` for a model instance by creating a context variable. One can iterate over the `form` object to get individual fields. This gives you fine-grain control over the form:

```
{% for field in form %}
{% ifequal field.name "comment" %}
  <!-- Customize the "comment" field, say, make CSS changes -->
...
{% endfor %}
```

But let's look at a simple example:

```
{% get_comment_form for entry as form %}
<!-- A context variable called form is created with the necessary hidden
fields, timestamps and security hashes -->
<table>
```

```
  <form action="{% comment_form_target %}" method="post">
    {% csrf_token %}
    {{ form }}
    <tr>
      <td colspan="2">
        <input type="submit" name="submit" value="Post">
        <input type="submit" name="preview" value="Preview">
      </td>
    </tr>
  </form>
</table>
```

Flagging If you want your users to be able to flag comments (say for profanity), you can just direct them (by placing a link in your comment list) to `/flag/{{ comment.id }}/`. Similarly, a user with requisite permissions (`"Can moderate comments"`) can approve and delete comments. This can also be done through the `admin` as you'll see later. You might also want to customize the following templates:

- `flag.html`
- `flagged.html`
- `approve.html`
- `approved.html`
- `delete.html`
- `deleted.html`

found under the directory structure we saw for `form.html`.

Feeds Suppose you want to export a *feed* of the latest comments, you can use the built-in `LatestCommentFeed`. Just enable it in your project's `urls.py`:

```
from django.conf.urls import patterns, url, include
from django.contrib.comments.feeds import LatestCommentFeed

urlpatterns = patterns('',
# ...
    (r'^feeds/latest/$', LatestCommentFeed()),
# ...
)
```

Now you should have the latest comment feeds being served off `/feeds/latest/`. Changed in version 1.3: *Please see the release notes* Prior to Django 1.3, the LatestCommentFeed was deployed using the syndication feed view:

```
from django.conf.urls import patterns
from django.contrib.comments.feeds import LatestCommentFeed

feeds = {
    'latest': LatestCommentFeed,
}

urlpatterns = patterns('',
# ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
# ...
)
```

Moderation Now that we have the comments framework working, we might want to have some moderation setup to administer the comments. The comments framework comes built-in with *generic comment moderation*. The comment moderation has the following features (all of which or only certain can be enabled):

- Enable comments for a particular model instance.
- Close comments after a particular (user-defined) number of days.
- Email new comments to the site-staff.

To enable comment moderation, we subclass the `CommentModerator` and register it with the moderation features we want. Let's suppose we want to close comments after 7 days of posting and also send out an email to the site staff. In `blog/models.py`, we register a comment moderator in the following way:

```
from django.contrib.comments.moderation import CommentModerator, moderator
from django.db import models

class Post(models.Model):
    title   = models.CharField(max_length = 255)
    content = models.TextField()
    posted_date = models.DateTimeField()

class PostModerator(CommentModerator):
    email_notification = True
    auto_close_field   = 'posted_date'
    # Close the comments after 7 days.
    close_after        = 7

moderator.register(Post, PostModerator)
```

The generic comment moderation also has the facility to remove comments. These comments can then be moderated by any user who has access to the `admin` site and the `Can moderate comments` permission (can be set under the `Users` page in the `admin`).

The moderator can `Flag`, `Approve` or `Remove` comments using the `Action` drop-down in the `admin` under the `Comments` page.

Note: Only a super-user will be able to delete comments from the database. `Remove Comments` only sets the `is_public` attribute to `False`.

6.4.4 The contenttypes framework

Django includes a `contenttypes` application that can track all of the models installed in your Django-powered project, providing a high-level, generic interface for working with your models.

Overview

At the heart of the contenttypes application is the `ContentType` model, which lives at `django.contrib.contenttypes.models.ContentType`. Instances of `ContentType` represent and store information about the models installed in your project, and new instances of `ContentType` are automatically created whenever new models are installed.

Instances of `ContentType` have methods for returning the model classes they represent and for querying objects from those models. `ContentType` also has a *custom manager* that adds methods for working with `ContentType` and for obtaining instances of `ContentType` for a particular model.

Relations between your models and `ContentType` can also be used to enable "generic" relationships between an instance of one of your models and instances of any model you have installed.

Installing the contenttypes framework

The contenttypes framework is included in the default `INSTALLED_APPS` list created by `django-admin.py startproject`, but if you've removed it or if you manually set up your `INSTALLED_APPS` list, you can enable it by adding `'django.contrib.contenttypes'` to your `INSTALLED_APPS` setting.

It's generally a good idea to have the contenttypes framework installed; several of Django's other bundled applications require it:

- The admin application uses it to log the history of each object added or changed through the admin interface.
- Django's `authentication framework` uses it to tie user permissions to specific models.
- Django's comments system (`django.contrib.comments`) uses it to "attach" comments to any installed model.

The `ContentType` model

class `ContentType`

Each instance of `ContentType` has three fields which, taken together, uniquely describe an installed model:

`app_label`

The name of the application the model is part of. This is taken from the `app_label` attribute of the model, and includes only the *last* part of the application's Python import path; "django.contrib.contenttypes", for example, becomes an `app_label` of "contenttypes".

`model`

The name of the model class.

`name`

The human-readable name of the model. This is taken from the `verbose_name` attribute of the model.

Let's look at an example to see how this works. If you already have the `contenttypes` application installed, and then add `the sites application` to your `INSTALLED_APPS` setting and run `manage.py syncdb` to install it, the model `django.contrib.sites.models.Site` will be installed into your database. Along with it a new instance of `ContentType` will be created with the following values:

- `app_label` will be set to `'sites'` (the last part of the Python path "django.contrib.sites").
- `model` will be set to `'site'`.
- `name` will be set to `'site'`.

Methods on `ContentType` instances

Each `ContentType` instance has methods that allow you to get from a `ContentType` instance to the model it represents, or to retrieve objects from that model:

`ContentType.`**`get_object_for_this_type`**`(`*`**kwargs`*`)`

Takes a set of valid *lookup arguments* for the model the `ContentType` represents, and does `a get() lookup` on that model, returning the corresponding object.

`ContentType.`**`model_class`**`()`

Returns the model class represented by this `ContentType` instance.

For example, we could look up the `ContentType` for the `User` model:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> user_type = ContentType.objects.get(app_label="auth", model="user")
>>> user_type
<ContentType: user>
```

And then use it to query for a particular `User`, or to get access to the `User` model class:

```
>>> user_type.model_class()
<class 'django.contrib.auth.models.User'>
>>> user_type.get_object_for_this_type(username='Guido')
<User: Guido>
```

Together, `get_object_for_this_type()` and `model_class()` enable two extremely important use cases:

1. Using these methods, you can write high-level generic code that performs queries on any installed model – instead of importing and using a single specific model class, you can pass an `app_label` and `model` into a `ContentType` lookup at runtime, and then work with the model class or retrieve objects from it.
2. You can relate another model to `ContentType` as a way of tying instances of it to particular model classes, and use these methods to get access to those model classes.

Several of Django's bundled applications make use of the latter technique. For example, `the permissions system` in Django's authentication framework uses a `Permission` model with a foreign key to `ContentType`; this lets `Permission` represent concepts like "can add blog entry" or "can delete news story".

The `ContentTypeManager`

class `ContentTypeManager`

`ContentType` also has a custom manager, `ContentTypeManager`, which adds the following methods:

`clear_cache`()

Clears an internal cache used by `ContentType` to keep track of which models for which it has created `ContentType` instances. You probably won't ever need to call this method yourself; Django will call it automatically when it's needed.

`get_for_model`(*model*[, *for_concrete_model=True*])

Takes either a model class or an instance of a model, and returns the `ContentType` instance representing that model.

`get_for_models`(**models*[, *for_concrete_models=True*])

Takes a variadic number of model classes, and returns a dictionary mapping the model classes to the `ContentType` instances representing them.

`get_by_natural_key`(*app_label*, *model*)

Returns the `ContentType` instance uniquely identified by the given application label and model name. The primary purpose of this method is to allow `ContentType` objects to be referenced via a *natural key* during deserialization.

The `get_for_model()` method is especially useful when you know you need to work with a `ContentType` but don't want to go to the trouble of obtaining the model's metadata to perform a manual lookup:

```
>>> from django.contrib.auth.models import User
>>> user_type = ContentType.objects.get_for_model(User)
>>> user_type
<ContentType: user>
```

New in version 1.5: *Please see the release notes* Prior to Django 1.5 `get_for_model()` and `get_for_models()` always returned the `ContentType` associated with the concrete model of the specified one(s). That means there was no way to retreive the `ContentType` of a proxy model using those methods. As of Django 1.5 you can now pass a boolean flag – respectively `for_concrete_model` and `for_concrete_models` – to specify wether or not you want to retreive the `ContentType` for the concrete or direct model.

Generic relations

Adding a foreign key from one of your own models to `ContentType` allows your model to effectively tie itself to another model class, as in the example of the `Permission` model above. But it's possible to go one step further and use `ContentType` to enable truly generic (sometimes called "polymorphic") relationships between models.

A simple example is a tagging system, which might look like this:

```
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes import generic

class TaggedItem(models.Model):
    tag = models.SlugField()
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey('content_type', 'object_id')

    def __unicode__(self):
        return self.tag
```

A normal `ForeignKey` can only "point to" one other model, which means that if the `TaggedItem` model used a `ForeignKey` it would have to choose one and only one model to store tags for. The contenttypes application provides a special field type which works around this and allows the relationship to be with any model:

class GenericForeignKey

There are three parts to setting up a `GenericForeignKey`:

1. Give your model a `ForeignKey` to `ContentType`.
2. Give your model a field that can store primary key values from the models you'll be relating to. For most models, this means a `PositiveIntegerField`. The usual name for this field is "object_id".
3. Give your model a `GenericForeignKey`, and pass it the names of the two fields described above. If these fields are named "content_type" and "object_id", you can omit this – those are the default field names `GenericForeignKey` will look for.

Primary key type compatibility

The "object_id" field doesn't have to be the same type as the primary key fields on the related models, but their primary key values must be coercible to the same type as the "object_id" field by its `get_db_prep_value()` method.

For example, if you want to allow generic relations to models with either `IntegerField` or `CharField` primary key fields, you can use `CharField` for the "object_id" field on your model since integers can be coerced to strings by `get_db_prep_value()`.

For maximum flexibility you can use a `TextField` which doesn't have a maximum length defined, however this may incur significant performance penalties depending on your database backend.

There is no one-size-fits-all solution for which field type is best. You should evaluate the models you expect to be pointing to and determine which solution will be most effective for your use case.

Serializing references to `ContentType` objects

If you're serializing data (for example, when generating `fixtures`) from a model that implements generic relations, you should probably be using a natural key to uniquely identify related `ContentType` objects. See *natural keys* and `dumpdata --natural` for more information.

This will enable an API similar to the one used for a normal `ForeignKey`; each `TaggedItem` will have a `content_object` field that returns the object it's related to, and you can also assign to that field or use it when creating a `TaggedItem`:

```
>>> from django.contrib.auth.models import User
>>> guido = User.objects.get(username='Guido')
>>> t = TaggedItem(content_object=guido, tag='bdfl')
>>> t.save()
>>> t.content_object
<User: Guido>
```

Due to the way `GenericForeignKey` is implemented, you cannot use such fields directly with filters (`filter()` and `exclude()`, for example) via the database API. Because a `GenericForeignKey` isn't a normal field object, these examples will *not* work:

```
# This will fail
>>> TaggedItem.objects.filter(content_object=guido)
# This will also fail
>>> TaggedItem.objects.get(content_object=guido)
```

Reverse generic relations

class `GenericRelation`

If you know which models you'll be using most often, you can also add a "reverse" generic relationship to enable an additional API. For example:

```
class Bookmark(models.Model):
    url = models.URLField()
    tags = generic.GenericRelation(TaggedItem)
```

`Bookmark` instances will each have a `tags` attribute, which can be used to retrieve their associated `TaggedItems`:

```
>>> b = Bookmark(url='https://www.djangoproject.com/')
>>> b.save()
>>> t1 = TaggedItem(content_object=b, tag='django')
>>> t1.save()
>>> t2 = TaggedItem(content_object=b, tag='python')
>>> t2.save()
>>> b.tags.all()
[<TaggedItem: django>, <TaggedItem: python>]
```

Just as `GenericForeignKey` accepts the names of the content-type and object-ID fields as arguments, so too does `GenericRelation`; if the model which has the generic foreign key is using non-default names for those fields, you must pass the names of the fields when setting up a `GenericRelation` to it. For example, if the `TaggedItem` model referred to above used fields named `content_type_fk` and `object_primary_key` to create its generic foreign key, then a `GenericRelation` back to it would need to be defined like so:

```
tags = generic.GenericRelation(TaggedItem,
                               content_type_field='content_type_fk',
                               object_id_field='object_primary_key')
```

Of course, if you don't add the reverse relationship, you can do the same types of lookups manually:

```
>>> b = Bookmark.objects.get(url='https://www.djangoproject.com/')
>>> bookmark_type = ContentType.objects.get_for_model(b)
>>> TaggedItem.objects.filter(content_type__pk=bookmark_type.id,
...                           object_id=b.id)
[<TaggedItem: django>, <TaggedItem: python>]
```

Note that if the model in a `GenericRelation` uses a non-default value for `ct_field` or `fk_field` in its `GenericForeignKey` (e.g. the `django.contrib.comments` app uses `ct_field="object_pk"`), you'll need to set `content_type_field` and/or `object_id_field` in the `GenericRelation` to match the `ct_field` and `fk_field`, respectively, in the `GenericForeignKey`:

```
comments = generic.GenericRelation(Comment, object_id_field="object_pk")
```

Note also, that if you delete an object that has a `GenericRelation`, any objects which have a `GenericForeignKey` pointing at it will be deleted as well. In the example above, this means that if a `Bookmark` object were deleted, any `TaggedItem` objects pointing at it would be deleted at the same time. New in version 1.3: *Please see the release notes* Unlike `ForeignKey`, `GenericForeignKey` does not accept an `on_delete` argument to customize this behavior; if desired, you can avoid the cascade-deletion simply by not using `GenericRelation`, and alternate behavior can be provided via the `pre_delete` signal.

Generic relations and aggregation

Django's database aggregation API doesn't work with a `GenericRelation`. For example, you might be tempted to try something like:

```
Bookmark.objects.aggregate(Count('tags'))
```

This will not work correctly, however. The generic relation adds extra filters to the queryset to ensure the correct content type, but the `aggregate()` method doesn't take them into account. For now, if you need aggregates on generic relations, you'll need to calculate them without using the aggregation API.

Generic relations in forms and admin

The `django.contrib.contenttypes.generic` module provides `BaseGenericInlineFormSet`, `GenericTabularInline` and `GenericStackedInline` (the last two are subclasses of `GenericInlineModelAdmin`). This enables the use of generic relations in forms and the admin. See the *model formset* and *admin* documentation for more information.

class `GenericInlineModelAdmin`

The `GenericInlineModelAdmin` class inherits all properties from an `InlineModelAdmin` class. However, it adds a couple of its own for working with the generic relation:

`ct_field`

The name of the `ContentType` foreign key field on the model. Defaults to `content_type`.

`ct_fk_field`

The name of the integer field that represents the ID of the related object. Defaults to `object_id`.

6.4.5 Cross Site Request Forgery protection

The CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries. This type of attack occurs when a malicious Web site contains a link, a form button or some javascript that is intended to perform some action on your Web site, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, 'login CSRF', where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered.

The first defense against CSRF attacks is to ensure that GET requests (and other 'safe' methods, as defined by 9.1.1 Safe Methods, HTTP 1.1, **RFC 2616**) are side-effect free. Requests via 'unsafe' methods, such as POST, PUT and DELETE, can then be protected by following the steps below.

How to use it

To enable CSRF protection for your views, follow these steps:

1. Add the middleware `'django.middleware.csrf.CsrfViewMiddleware'` to your list of middleware classes, `MIDDLEWARE_CLASSES`. (It should come before any view middleware that assume that CSRF attacks have been dealt with.)

   Alternatively, you can use the decorator `csrf_protect()` on particular views you want to protect (see below).

2. In any template that uses a POST form, use the `csrf_token` tag inside the `<form>` element if the form is for an internal URL, e.g.:

   ```
   <form action="." method="post">{% csrf_token %}
   ```

   This should not be done for POST forms that target external URLs, since that would cause the CSRF token to be leaked, leading to a vulnerability.

3. In the corresponding view functions, ensure that the `'django.core.context_processors.csrf'` context processor is being used. Usually, this can be done in one of two ways:

   (a) Use RequestContext, which always uses `'django.core.context_processors.csrf'` (no matter what your TEMPLATE_CONTEXT_PROCESSORS setting). If you are using generic views or contrib apps, you are covered already, since these apps use RequestContext throughout.

   (b) Manually import and use the processor to generate the CSRF token and add it to the template context. e.g.:

   ```
   from django.core.context_processors import csrf
   from django.shortcuts import render_to_response

   def my_view(request):
       c = {}
       c.update(csrf(request))
       # ... view code here
       return render_to_response("a_template.html", c)
   ```

   You may want to write your own `render_to_response()` wrapper that takes care of this step for you.

The utility script `extras/csrf_migration_helper.py` can help to automate the finding of code and templates that may need these steps. It contains full help on how to use it.

AJAX

While the above method can be used for AJAX POST requests, it has some inconveniences: you have to remember to pass the CSRF token in as POST data with every POST request. For this reason, there is an alternative method: on each XMLHttpRequest, set a custom *X-CSRFToken* header to the value of the CSRF token. This is often easier, because many javascript frameworks provide hooks that allow headers to be set on every request. In jQuery, you can use the `ajaxSend` event as follows:

```
jQuery(document).ajaxSend(function(event, xhr, settings) {
    function getCookie(name) {
        var cookieValue = null;
        if (document.cookie && document.cookie != '') {
            var cookies = document.cookie.split(';');
            for (var i = 0; i < cookies.length; i++) {
                var cookie = jQuery.trim(cookies[i]);
                // Does this cookie string begin with the name we want?
                if (cookie.substring(0, name.length + 1) == (name + '=')) {
                    cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                    break;
                }
            }
        }
        return cookieValue;
    }
    function sameOrigin(url) {
        // url could be relative or scheme relative or absolute
        var host = document.location.host; // host + port
        var protocol = document.location.protocol;
        var sr_origin = '//' + host;
        var origin = protocol + sr_origin;
        // Allow absolute or scheme relative URLs to same origin
        return (url == origin || url.slice(0, origin.length + 1) == origin + '/') ||
            (url == sr_origin || url.slice(0, sr_origin.length + 1) == sr_origin + '/') ||
            // or any other URL that isn't scheme relative or absolute i.e relative.
            !(/^(\/\/|http:|https:).*/.test(url));
    }
    function safeMethod(method) {
        return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
    }

    if (!safeMethod(settings.type) && sameOrigin(settings.url)) {
        xhr.setRequestHeader("X-CSRFToken", getCookie('csrftoken'));
    }
});
```

Note: Due to a bug introduced in jQuery 1.5, the example above will not work correctly on that version. Make sure you are running at least jQuery 1.5.1.

Adding this to a javascript file that is included on your site will ensure that AJAX POST requests that are made via jQuery will not be caught by the CSRF protection.

The above code could be simplified by using the jQuery cookie plugin to replace `getCookie`, and settings.crossDomain in jQuery 1.5 and later to replace `sameOrigin`.

In addition, if the CSRF cookie has not been sent to the client by use of `csrf_token`, you may need to ensure the client receives the cookie by using `ensure_csrf_cookie()`.

Other template engines

When using a different template engine than Django's built-in engine, you can set the token in your forms manually after making sure it's available in the template context.

For example, in the Cheetah template language, your form could contain the following:

```
<div style="display:none">
    <input type="hidden" name="csrfmiddlewaretoken" value="$csrf_token"/>
</div>
```

You can use JavaScript similar to the *AJAX code* above to get the value of the CSRF token.

The decorator method

Rather than adding `CsrfViewMiddleware` as a blanket protection, you can use the `csrf_protect` decorator, which has exactly the same functionality, on particular views that need the protection. It must be used **both** on views that insert the CSRF token in the output, and on those that accept the POST form data. (These are often the same view function, but not always).

Use of the decorator by itself is **not recommended**, since if you forget to use it, you will have a security hole. The 'belt and braces' strategy of using both is fine, and will incur minimal overhead.

csrf_protect(*view*)

Decorator that provides the protection of `CsrfViewMiddleware` to a view.

Usage:

```
from django.views.decorators.csrf import csrf_protect
from django.shortcuts import render

@csrf_protect
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

Rejected requests

By default, a '403 Forbidden' response is sent to the user if an incoming request fails the checks performed by `CsrfViewMiddleware`. This should usually only be seen when there is a genuine Cross Site Request Forgery, or when, due to a programming error, the CSRF token has not been included with a POST form.

The error page, however, is not very friendly, so you may want to provide your own view for handling this condition. To do this, simply set the `CSRF_FAILURE_VIEW` setting.

How it works

The CSRF protection is based on the following things:

1. A CSRF cookie that is set to a random value (a session independent nonce, as it is called), which other sites will not have access to.

   This cookie is set by `CsrfViewMiddleware`. It is meant to be permanent, but since there is no way to set a cookie that never expires, it is sent with every response that has called `django.middleware.csrf.get_token()` (the function used internally to retrieve the CSRF token).

2. A hidden form field with the name 'csrfmiddlewaretoken' present in all outgoing POST forms. The value of this field is the value of the CSRF cookie.

   This part is done by the template tag.

3. For all incoming requests that are not using HTTP GET, HEAD, OPTIONS or TRACE, a CSRF cookie must be present, and the 'csrfmiddlewaretoken' field must be present and correct. If it isn't, the user will get a 403 error.

   This check is done by `CsrfViewMiddleware`.

4. In addition, for HTTPS requests, strict referer checking is done by `CsrfViewMiddleware`. This is necessary to address a Man-In-The-Middle attack that is possible under HTTPS when using a session independent nonce, due to the fact that HTTP 'Set-Cookie' headers are (unfortunately) accepted by clients that are talking to a site under HTTPS. (Referer checking is not done for HTTP requests because the presence of the Referer header is not reliable enough under HTTP.)

This ensures that only forms that have originated from your Web site can be used to POST data back.

It deliberately ignores GET requests (and other requests that are defined as 'safe' by **RFC 2616**). These requests ought never to have any potentially dangerous side effects , and so a CSRF attack with a GET request ought to be harmless. **RFC 2616** defines POST, PUT and DELETE as 'unsafe', and all other methods are assumed to be unsafe, for maximum protection.

Caching

If the `csrf_token` template tag is used by a template (or the `get_token` function is called some other way), `CsrfViewMiddleware` will add a cookie and a `Vary: Cookie` header to the response. This means that the middleware will play well with the cache middleware if it is used as instructed (`UpdateCacheMiddleware` goes before all other middleware).

However, if you use cache decorators on individual views, the CSRF middleware will not yet have been able to set the Vary header or the CSRF cookie, and the response will be cached without either one. In this case, on any views that will require a CSRF token to be inserted you should use the `django.views.decorators.csrf.csrf_protect()` decorator first:

```
from django.views.decorators.cache import cache_page
from django.views.decorators.csrf import csrf_protect

@cache_page(60 * 15)
@csrf_protect
def my_view(request):
    # ...
```

Testing

The `CsrfViewMiddleware` will usually be a big hindrance to testing view functions, due to the need for the CSRF token which must be sent with every POST request. For this reason, Django's HTTP client for tests has been modified to set a flag on requests which relaxes the middleware and the `csrf_protect` decorator so that they no longer rejects requests. In every other respect (e.g. sending cookies etc.), they behave the same.

If, for some reason, you *want* the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks:

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

Limitations

Subdomains within a site will be able to set cookies on the client for the whole domain. By setting the cookie and using a corresponding token, subdomains will be able to circumvent the CSRF protection. The only way to avoid this is to ensure that subdomains are controlled by trusted users (or, are at least unable to set cookies). Note that even without CSRF, there are other vulnerabilities, such as session fixation, that make giving subdomains to untrusted parties a bad idea, and these vulnerabilities cannot easily be fixed with current browsers.

Edge cases

Certain views can have unusual requirements that mean they don't fit the normal pattern envisaged here. A number of utilities can be useful in these situations. The scenarios they might be needed in are described in the following section.

Utilities

csrf_exempt(*view*)
This decorator marks a view as being exempt from the protection ensured by the middleware. Example:

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def my_view(request):
    return HttpResponse('Hello world')
```

requires_csrf_token(*view*)
Normally the `csrf_token` template tag will not work if `CsrfViewMiddleware.process_view` or an equivalent like `csrf_protect` has not run. The view decorator `requires_csrf_token` can be used to ensure the template tag does work. This decorator works similarly to `csrf_protect`, but never rejects an incoming request.

Example:

```
from django.views.decorators.csrf import requires_csrf_token
from django.shortcuts import render

@requires_csrf_token
def my_view(request):
    c = {}
    # ...
    return render(request, "a_template.html", c)
```

ensure_csrf_cookie(*view*)
New in version 1.4: *Please see the release notes* This decorator forces a view to send the CSRF cookie.

Scenarios

CSRF protection should be disabled for just a few views Most views requires CSRF protection, but a few do not.

Solution: rather than disabling the middleware and applying `csrf_protect` to all the views that need it, enable the middleware and use `csrf_exempt()`.

CsrfViewMiddleware.process_view not used There are cases when `CsrfViewMiddleware.process_view` may not have run before your view is run - 404 and 500 handlers, for example - but you still need the CSRF token in a form.

Solution: use `requires_csrf_token()`

Unprotected view needs the CSRF token There may be some views that are unprotected and have been exempted by `csrf_exempt`, but still need to include the CSRF token.

Solution: use `csrf_exempt()` followed by `requires_csrf_token()`. (i.e. `requires_csrf_token` should be the innermost decorator).

View needs protection for one path A view needs CRSF protection under one set of conditions only, and mustn't have it for the rest of the time.

Solution: use `csrf_exempt()` for the whole view function, and `csrf_protect()` for the path within it that needs protection. Example:

```
from django.views.decorators.csrf import csrf_exempt, csrf_protect

@csrf_exempt
def my_view(request):

    @csrf_protect
    def protected_path(request):
        do_something()

    if some_condition():
       return protected_path(request)
    else:
       do_something_else()
```

Page uses AJAX without any HTML form A page makes a POST request via AJAX, and the page does not have an HTML form with a `csrf_token` that would cause the required CSRF cookie to be sent.

Solution: use `ensure_csrf_cookie()` on the view that sends the page.

Contrib and reusable apps

Because it is possible for the developer to turn off the `CsrfViewMiddleware`, all relevant views in contrib apps use the `csrf_protect` decorator to ensure the security of these applications against CSRF. It is recommended that the developers of other reusable apps that want the same guarantees also use the `csrf_protect` decorator on their views.

Settings

A number of settings can be used to control Django's CSRF behavior.

CSRF_COOKIE_DOMAIN

Default: `None`

The domain to be used when setting the CSRF cookie. This can be useful for easily allowing cross-subdomain requests to be excluded from the normal cross site request forgery protection. It should be set to a string such as `".example.com"` to allow a POST request from a form on one subdomain to be accepted by a view served from another subdomain.

Please note that, with or without use of this setting, this CSRF protection mechanism is not safe against cross-subdomain attacks – see Limitations.

CSRF_COOKIE_NAME

Default: `'csrftoken'`

The name of the cookie to use for the CSRF authentication token. This can be whatever you want.

CSRF_COOKIE_PATH

New in version 1.4: *Please see the release notes* Default: `'/'`

The path set on the CSRF cookie. This should either match the URL path of your Django installation or be a parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths, and each instance will only see its own CSRF cookie.

CSRF_COOKIE_SECURE

New in version 1.4: *Please see the release notes* Default: `False`

Whether to use a secure cookie for the CSRF cookie. If this is set to `True`, the cookie will be marked as "secure," which means browsers may ensure that the cookie is only sent under an HTTPS connection.

CSRF_FAILURE_VIEW

Default: `'django.views.csrf.csrf_failure'`

A dotted path to the view function to be used when an incoming request is rejected by the CSRF protection. The function should have this signature:

```
def csrf_failure(request, reason="")
```

where `reason` is a short message (intended for developers or logging, not for end users) indicating the reason the request was rejected.

6.4.6 Databrowse

Deprecated since version 1.4: This module has been deprecated. Databrowse is a Django application that lets you browse your data.

As the Django admin dynamically creates an admin interface by introspecting your models, Databrowse dynamically creates a rich, browsable Web site by introspecting your models.

How to use Databrowse

1. Point Django at the default Databrowse templates. There are two ways to do this:
   - Add `'django.contrib.databrowse'` to your `INSTALLED_APPS` setting. This will work if your `TEMPLATE_LOADERS` setting includes the `app_directories` template loader (which is the case by default). See the *template loader docs* for more.

- Otherwise, determine the full filesystem path to the `django/contrib/databrowse/templates` directory, and add that directory to your `TEMPLATE_DIRS` setting.

2. Register a number of models with the Databrowse site:

```
from django.contrib import databrowse
from myapp.models import SomeModel, SomeOtherModel, YetAnotherModel

databrowse.site.register(SomeModel)
databrowse.site.register(SomeOtherModel, YetAnotherModel)
```

   Note that you should register the model *classes*, not instances. Changed in version 1.4: *Please see the release notes* Since Django 1.4, it is possible to register several models in the same call to `register()`.

   It doesn't matter where you put this, as long as it gets executed at some point. A good place for it is in your *URLconf file* (`urls.py`).

3. Change your URLconf to import the `databrowse` module:

```
from django.contrib import databrowse
```

   ...and add the following line to your URLconf:

```
(r'^databrowse/(.*)', databrowse.site.root),
```

   The prefix doesn't matter – you can use `databrowse/` or `db/` or whatever you'd like.

4. Run the Django server and visit `/databrowse/` in your browser.

Requiring user login

You can restrict access to logged-in users with only a few extra lines of code. Simply add the following import to your URLconf:

```
from django.contrib.auth.decorators import login_required
```

Then modify the *URLconf* so that the `databrowse.site.root()` view is decorated with `django.contrib.auth.decorators.login_required()`:

```
(r'^databrowse/(.*)', login_required(databrowse.site.root)),
```

If you haven't already added support for user logins to your *URLconf*, as described in the *user authentication docs*, then you will need to do so now with the following mapping:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login'),
```

The final step is to create the login form required by `django.contrib.auth.views.login()`. The *user authentication docs* provide full details and a sample template that can be used for this purpose.

6.4.7 The flatpages app

Django comes with an optional "flatpages" application. It lets you store simple "flat" HTML content in a database and handles the management for you via Django's admin interface and a Python API.

A flatpage is a simple object with a URL, title and content. Use it for one-off, special-case pages, such as "About" or "Privacy Policy" pages, that you want to store in a database but for which you don't want to develop a custom Django application.

A flatpage can use a custom template or a default, systemwide flatpage template. It can be associated with one, or multiple, sites.

The content field may optionally be left blank if you prefer to put your content in a custom template.

Here are some examples of flatpages on Django-powered sites:

- http://www.lawrence.com/about/contact/
- http://www2.ljworld.com/site/rules/

Installation

To install the flatpages app, follow these steps:

1. Install the `sites framework` by adding `'django.contrib.sites'` to your `INSTALLED_APPS` setting, if it's not already in there.

   Also make sure you've correctly set `SITE_ID` to the ID of the site the settings file represents. This will usually be `1` (i.e. `SITE_ID = 1`, but if you're using the sites framework to manage multiple sites, it could be the ID of a different site.

2. Add `'django.contrib.flatpages'` to your `INSTALLED_APPS` setting.

3. Add `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` to your `MIDDLEWARE_CLASSES` setting.

4. Run the command `manage.py syncdb`.

How it works

`manage.py syncdb` creates two tables in your database: `django_flatpage` and `django_flatpage_sites`. `django_flatpage` is a simple lookup table that simply maps a URL to a title and bunch of text content. `django_flatpage_sites` associates a flatpage with a site.

The `FlatpageFallbackMiddleware` does all of the work.

class `FlatpageFallbackMiddleware`

Each time any Django application raises a 404 error, this middleware checks the flatpages database for the requested URL as a last resort. Specifically, it checks for a flatpage with the given URL with a site ID that corresponds to the `SITE_ID` setting.

If it finds a match, it follows this algorithm:

- If the flatpage has a custom template, it loads that template. Otherwise, it loads the template `flatpages/default.html`.
- It passes that template a single context variable, `flatpage`, which is the flatpage object. It uses `RequestContext` in rendering the template.

Changed in version 1.4: The middleware will only add a trailing slash and redirect (by looking at the `APPEND_SLASH` setting) if the resulting URL refers to a valid flatpage. Previously requesting a non-existent flatpage would redirect to the same URL with an apppended slash first and subsequently raise a 404.Changed in version 1.4: Redirects by the middleware are permanent (301 status code) instead of temporary (302) to match behavior of the `CommonMiddleware`. If it doesn't find a match, the request continues to be processed as usual.

The middleware only gets activated for 404s – not for 500s or responses of any other status code.

Flatpages will not apply view middleware

Because the `FlatpageFallbackMiddleware` is applied only after URL resolution has failed and produced a 404, the response it returns will not apply any *view middleware* methods. Only requests which are successfully routed to a view via normal URL resolution apply view middleware.

Note that the order of `MIDDLEWARE_CLASSES` matters. Generally, you can put `FlatpageFallbackMiddleware` at the end of the list. This means it will run first when processing the response, and ensures that any other response-processing middlewares see the real flatpage response rather than the 404.

For more on middleware, read the *middleware docs*.

Ensure that your 404 template works

Note that the `FlatpageFallbackMiddleware` only steps in once another view has successfully produced a 404 response. If another view or middleware class attempts to produce a 404 but ends up raising an exception instead (such as a `TemplateDoesNotExist` exception if your site does not have an appropriate template to use for HTTP 404 responses), the response will become an HTTP 500 ("Internal Server Error") and the `FlatpageFallbackMiddleware` will not attempt to serve a flat page.

How to add, change and delete flatpages

Via the admin interface

If you've activated the automatic Django admin interface, you should see a "Flatpages" section on the admin index page. Edit flatpages as you edit any other object in the system.

Via the Python API

class `FlatPage`

Flatpages are represented by a standard *Django model*, which lives in django/contrib/flatpages/models.py. You can access flatpage objects via the *Django database API*.

Check for duplicate flatpage URLs.

If you add or modify flatpages via your own code, you will likely want to check for duplicate flatpage URLs within the same site. The flatpage form used in the admin performs this validation check, and can be imported from `django.contrib.flatpages.forms.FlatPageForm` and used in your own views.

Flatpage templates

By default, flatpages are rendered via the template `flatpages/default.html`, but you can override that for a particular flatpage: in the admin, a collapsed fieldset titled "Advanced options" (clicking will expand it) contains a field for specifying a template name. If you're creating a flat page via the Python API you can simply set the template name as the field `template_name` on the `FlatPage` object.

Creating the `flatpages/default.html` template is your responsibility; in your template directory, just create a `flatpages` directory containing a file `default.html`.

Flatpage templates are passed a single context variable, `flatpage`, which is the flatpage object.

Here's a sample `flatpages/default.html` template:

```
<!DOCTYPE html>
<html>
<head>
<title>{{ flatpage.title }}</title>
</head>
<body>
{{ flatpage.content }}
</body>
</html>
```

Since you're already entering raw HTML into the admin page for a flatpage, both `flatpage.title` and `flatpage.content` are marked as **not** requiring *automatic HTML escaping* in the template.

Getting a list of `FlatPage` objects in your templates

New in version 1.3: *Please see the release notes* The flatpages app provides a template tag that allows you to iterate over all of the available flatpages on the *current site*.

Like all custom template tags, you'll need to *load its custom tag library* before you can use it. After loading the library, you can retrieve all current flatpages via the `get_flatpages` tag:

```
{% load flatpages %}
{% get_flatpages as flatpages %}
<ul>
    {% for page in flatpages %}
        <li><a href="{{ page.url }}">{{ page.title }}</a></li>
    {% endfor %}
</ul>
```

Displaying `registration_required` flatpages

By default, the `get_flatpages` templatetag will only show flatpages that are marked `registration_required = False`. If you want to display registration-protected flatpages, you need to specify an authenticated user using a''for'' clause.

For example:

```
{% get_flatpages for someuser as about_pages %}
```

If you provide an anonymous user, `get_flatpages` will behave the same as if you hadn't provided a user – i.e., it will only show you public flatpages.

Limiting flatpages by base URL

An optional argument, `starts_with`, can be applied to limit the returned pages to those beginning with a particular base URL. This argument may be passed as a string, or as a variable to be resolved from the context.

For example:

```
{% get_flatpages '/about/' as about_pages %}
{% get_flatpages about_prefix as about_pages %}
{% get_flatpages '/about/' for someuser as about_pages %}
```

6.4.8 django.contrib.formtools

A set of high-level abstractions for Django forms (`django.forms`).

Form preview

Django comes with an optional "form preview" application that helps automate the following workflow:

"Display an HTML form, force a preview, then do something with the submission."

To force a preview of a form submission, all you have to do is write a short Python class.

Overview

Given a `django.forms.Form` subclass that you define, this application takes care of the following workflow:

1. Displays the form as HTML on a Web page.
2. Validates the form data when it's submitted via POST. a. If it's valid, displays a preview page. b. If it's not valid, redisplays the form with error messages.
3. When the "confirmation" form is submitted from the preview page, calls a hook that you define – a `done()` method that gets passed the valid data.

The framework enforces the required preview by passing a shared-secret hash to the preview page via hidden form fields. If somebody tweaks the form parameters on the preview page, the form submission will fail the hash-comparison test.

How to use `FormPreview`

1. Point Django at the default FormPreview templates. There are two ways to do this:
   - Add `'django.contrib.formtools'` to your `INSTALLED_APPS` setting. This will work if your `TEMPLATE_LOADERS` setting includes the `app_directories` template loader (which is the case by default). See the *template loader docs* for more.
   - Otherwise, determine the full filesystem path to the `django/contrib/formtools/templates` directory, and add that directory to your `TEMPLATE_DIRS` setting.
2. Create a `FormPreview` subclass that overrides the `done()` method:

   ```
   from django.contrib.formtools.preview import FormPreview
   from myapp.models import SomeModel

   class SomeModelFormPreview(FormPreview):

       def done(self, request, cleaned_data):
           # Do something with the cleaned_data, then redirect
           # to a "success" page.
           return HttpResponseRedirect('/form/success')
   ```

   This method takes an `HttpRequest` object and a dictionary of the form data after it has been validated and cleaned. It should return an `HttpResponseRedirect` that is the end result of the form being submitted.
3. Change your URLconf to point to an instance of your `FormPreview` subclass:

```
from myapp.preview import SomeModelFormPreview
from myapp.forms import SomeModelForm
from django import forms
```

...and add the following line to the appropriate model in your URLconf:

```
(r'^post/$', SomeModelFormPreview(SomeModelForm)),
```

where `SomeModelForm` is a Form or ModelForm class for the model.

4. Run the Django server and visit `/post/` in your browser.

`FormPreview` classes

class `FormPreview`

A `FormPreview` class is a simple Python class that represents the preview workflow. `FormPreview` classes must subclass `django.contrib.formtools.preview.FormPreview` and override the `done()` method. They can live anywhere in your codebase.

`FormPreview` templates

By default, the form is rendered via the template `formtools/form.html`, and the preview page is rendered via the template `formtools/preview.html`. These values can be overridden for a particular form preview by setting `preview_template` and `form_template` attributes on the FormPreview subclass. See `django/contrib/formtools/templates` for the default templates.

Advanced `FormPreview` methods

`FormPreview.`**`process_preview`**`()`

Given a validated form, performs any extra processing before displaying the preview page, and saves any extra data in context.

By default, this method is empty. It is called after the form is validated, but before the context is modified with hash information and rendered.

Form wizard

Django comes with an optional "form wizard" application that splits *forms* across multiple Web pages. It maintains state in one of the backends so that the full server-side processing can be delayed until the submission of the final form.

You might want to use this if you have a lengthy form that would be too unwieldy for display on a single page. The first page might ask the user for core information, the second page might ask for less important information, etc.

The term "wizard", in this context, is explained on Wikipedia.

How it works

Here's the basic workflow for how a user would use a wizard:

1. The user visits the first page of the wizard, fills in the form and submits it.

2. The server validates the data. If it's invalid, the form is displayed again, with error messages. If it's valid, the server saves the current state of the wizard in the backend and redirects to the next step.
3. Step 1 and 2 repeat, for every subsequent form in the wizard.
4. Once the user has submitted all the forms and all the data has been validated, the wizard processes the data – saving it to the database, sending an email, or whatever the application needs to do.

Usage

This application handles as much machinery for you as possible. Generally, you just have to do these things:

1. Define a number of `Form` classes – one per wizard page.
2. Create a `WizardView` subclass that specifies what to do once all of your forms have been submitted and validated. This also lets you override some of the wizard's behavior.
3. Create some templates that render the forms. You can define a single, generic template to handle every one of the forms, or you can define a specific template for each form.
4. Add `django.contrib.formtools` to your `INSTALLED_APPS` list in your settings file.
5. Point your URLconf at your `WizardView` `as_view()` method.

Defining `Form` classes The first step in creating a form wizard is to create the `Form` classes. These should be standard `django.forms.Form` classes, covered in the *forms documentation*. These classes can live anywhere in your codebase, but convention is to put them in a file called `forms.py` in your application.

For example, let's write a "contact form" wizard, where the first page's form collects the sender's email address and subject, and the second page collects the message itself. Here's what the `forms.py` might look like:

```
from django import forms

class ContactForm1(forms.Form):
    subject = forms.CharField(max_length=100)
    sender = forms.EmailField()

class ContactForm2(forms.Form):
    message = forms.CharField(widget=forms.Textarea)
```

Note: In order to use `FileField` in any form, see the section *Handling files* below to learn more about what to do.

Creating a `WizardView` class The next step is to create a `django.contrib.formtools.wizard.views.WizardView` subclass. You can also use the `SessionWizardView` or `CookieWizardView` classes which preselect the backend used for storing information during execution of the wizard (as their names indicate, server-side sessions and browser cookies respectively).

Note: To use the `SessionWizardView` follow the instructions in the *sessions documentation* on how to enable sessions.

We will use the `SessionWizardView` in all examples but is is completely fine to use the `CookieWizardView` instead. As with your `Form` classes, this `WizardView` class can live anywhere in your codebase, but convention is to put it in `views.py`.

The only requirement on this subclass is that it implement a `done()` method.

`WizardView.done`(*form_list*)

This method specifies what should happen when the data for *every* form is submitted and validated. This method is passed a list of validated `Form` instances.

In this simplistic example, rather than performing any database operation, the method simply renders a template of the validated data:

```
from django.shortcuts import render_to_response
from django.contrib.formtools.wizard.views import SessionWizardView

class ContactWizard(SessionWizardView):
    def done(self, form_list, **kwargs):
        return render_to_response('done.html', {
            'form_data': [form.cleaned_data for form in form_list],
        })
```

Note that this method will be called via `POST`, so it really ought to be a good Web citizen and redirect after processing the data. Here's another example:

```
from django.http import HttpResponseRedirect
from django.contrib.formtools.wizard.views import SessionWizardView

class ContactWizard(SessionWizardView):
    def done(self, form_list, **kwargs):
        do_something_with_the_form_data(form_list)
        return HttpResponseRedirect('/page-to-redirect-to-when-done/')
```

See the section *Advanced WizardView methods* below to learn about more `WizardView` hooks.

Creating templates for the forms Next, you'll need to create a template that renders the wizard's forms. By default, every form uses a template called `formtools/wizard/wizard_form.html`. You can change this template name by overriding either the `template_name` attribute or the `get_template_names()` method, which are documented in the `TemplateResponseMixin` documentation. The latter one allows you to use a different template for each form.

This template expects a `wizard` object that has various items attached to it:

- `form` – The `Form` or `BaseFormSet` instance for the current step (either empty or with errors).
- `steps` – A helper object to access the various steps related data:
  - `step0` – The current step (zero-based).
  - `step1` – The current step (one-based).
  - `count` – The total number of steps.
  - `first` – The first step.
  - `last` – The last step.
  - `current` – The current (or first) step.
  - `next` – The next step.
  - `prev` – The previous step.
  - `index` – The index of the current step.
  - `all` – A list of all steps of the wizard.

You can supply additional context variables by using the `get_context_data()` method of your `WizardView` subclass.

Here's a full example template:

```
{% extends "base.html" %}
{% load i18n %}

{% block head %}
{{ wizard.form.media }}
{% endblock %}

{% block content %}
<p>Step {{ wizard.steps.step1 }} of {{ wizard.steps.count }}</p>
<form action="" method="post">{% csrf_token %}
<table>
{{ wizard.management_form }}
{% if wizard.form.forms %}
    {{ wizard.form.management_form }}
    {% for form in wizard.form.forms %}
        {{ form }}
    {% endfor %}
{% else %}
    {{ wizard.form }}
{% endif %}
</table>
{% if wizard.steps.prev %}
<button name="wizard_goto_step" type="submit" value="{{ wizard.steps.first }}">{% trans "first step"
<button name="wizard_goto_step" type="submit" value="{{ wizard.steps.prev }}">{% trans "prev step" %
{% endif %}
<input type="submit" value="{% trans "submit" %}"/>
</form>
{% endblock %}
```

Note: Note that `{{ wizard.management_form }}` **must be used** for the wizard to work properly.

Hooking the wizard into a URLconf Finally, we need to specify which forms to use in the wizard, and then deploy the new `WizardView` object at an URL in the `urls.py`. The wizard's `as_view()` method takes a list of your `Form` classes as an argument during instantiation:

```
from django.conf.urls import patterns

from myapp.forms import ContactForm1, ContactForm2
from myapp.views import ContactWizard

urlpatterns = patterns('',
    (r'^contact/$', ContactWizard.as_view([ContactForm1, ContactForm2])),
)
```

Advanced `WizardView` methods

class `WizardView`

Aside from the `done()` method, `WizardView` offers a few advanced method hooks that let you customize how your wizard works.

Some of these methods take an argument `step`, which is a zero-based counter as string representing the current step of the wizard. (E.g., the first form is `'0'` and the second form is `'1'`)

WizardView.**get_form_prefix**(*step*)
Given the step, returns a form prefix to use. By default, this simply uses the step itself. For more, see the *form prefix documentation*.

WizardView.**get_form_initial**(*step*)
Returns a dictionary which will be passed as the `initial` argument when instantiating the Form instance for step `step`. If no initial data was provided while initializing the form wizard, an empty dictionary should be returned.

The default implementation:

```
def get_form_initial(self, step):
    return self.initial_dict.get(step, {})
```

WizardView.**get_form_kwargs**(*step*)
Returns a dictionary which will be used as the keyword arguments when instantiating the form instance on given `step`.

The default implementation:

```
def get_form_kwargs(self, step):
    return {}
```

WizardView.**get_form_instance**(*step*)
This method will be called only if a `ModelForm` is used as the form for step `step`.

Returns an `Model` object which will be passed as the `instance` argument when instantiating the ModelForm for step `step`. If no instance object was provided while initializing the form wizard, `None` will be returned.

The default implementation:

```
def get_form_instance(self, step):
    return self.instance_dict.get(step, None)
```

WizardView.**get_context_data**(*form*, ***kwargs*)
Returns the template context for a step. You can overwrite this method to add more data for all or some steps. This method returns a dictionary containing the rendered form step.

The default template context variables are:

- Any extra data the storage backend has stored
- `form` – form instance of the current step
- `wizard` – the wizard instance itself

Example to add extra variables for a specific step:

```
def get_context_data(self, form, **kwargs):
    context = super(MyWizard, self).get_context_data(form=form, **kwargs)
    if self.steps.current == 'my_step_name':
        context.update({'another_var': True})
    return context
```

WizardView.**get_prefix**(**args*, ***kwargs*)
This method returns a prefix for use by the storage backends. Backends use the prefix as a mechanism to allow data to be stored separately for each wizard. This allows wizards to store their data in a single backend without overwriting each other.

You can change this method to make the wizard data prefix more unique to, e.g. have multiple instances of one wizard in one session.

Default implementation:

```
    def get_prefix(self, *args, **kwargs):
        # use the lowercase underscore version of the class name
        return normalize_name(self.__class__.__name__)
```

`WizardView.`**`get_form`**(*step=None*, *data=None*, *files=None*)

This method constructs the form for a given `step`. If no `step` is defined, the current step will be determined automatically. The method gets three arguments:

- `step` – The step for which the form instance should be generated.
- `data` – Gets passed to the form's data argument
- `files` – Gets passed to the form's files argument

You can override this method to add extra arguments to the form instance.

Example code to add a user attribute to the form on step 2:

```
def get_form(self, step=None, data=None, files=None):
    form = super(MyWizard, self).get_form(step, data, files)
    if step == '1':
        form.user = self.request.user
    return form
```

`WizardView.`**`process_step`**(*form*)

Hook for modifying the wizard's internal state, given a fully validated `Form` object. The Form is guaranteed to have clean, valid data.

This method gives you a way to post-process the form data before the data gets stored within the storage backend. By default it just returns the `form.data` dictionary. You should not manipulate the data here but you can use it to do some extra work if needed (e.g. set storage extra data).

Note that this method is called every time a page is rendered for *all* submitted steps.

The default implementation:

```
def process_step(self, form):
    return self.get_form_step_data(form)
```

`WizardView.`**`process_step_files`**(*form*)

This method gives you a way to post-process the form files before the files gets stored within the storage backend. By default it just returns the `form.files` dictionary. You should not manipulate the data here but you can use it to do some extra work if needed (e.g. set storage extra data).

Default implementation:

```
def process_step_files(self, form):
    return self.get_form_step_files(form)
```

`WizardView.`**`render_revalidation_failure`**(*step*, *form*, ***kwargs*)

When the wizard thinks all steps have passed it revalidates all forms with the data from the backend storage.

If any of the forms don't validate correctly, this method gets called. This method expects two arguments, `step` and `form`.

The default implementation resets the current step to the first failing form and redirects the user to the invalid form.

Default implementation:

```
def render_revalidation_failure(self, step, form, **kwargs):
    self.storage.current_step = step
    return self.render(form, **kwargs)
```

WizardView.**get_form_step_data**(*form*)

This method fetches the data from the `form` Form instance and returns the dictionary. You can use this method to manipulate the values before the data gets stored in the storage backend.

Default implementation:

```
def get_form_step_data(self, form):
    return form.data
```

WizardView.**get_form_step_files**(*form*)

This method returns the form files. You can use this method to manipulate the files before the data gets stored in the storage backend.

Default implementation:

```
def get_form_step_files(self, form):
    return form.files
```

WizardView.**render**(*form*, ***kwargs*)

This method gets called after the GET or POST request has been handled. You can hook in this method to, e.g. change the type of HTTP response.

Default implementation:

```
def render(self, form=None, **kwargs):
    form = form or self.get_form()
    context = self.get_context_data(form=form, **kwargs)
    return self.render_to_response(context)
```

Providing initial data for the forms

WizardView.**initial_dict**

Initial data for a wizard's `Form` objects can be provided using the optional `initial_dict` keyword argument. This argument should be a dictionary mapping the steps to dictionaries containing the initial data for each step. The dictionary of initial data will be passed along to the constructor of the step's `Form`:

```
>>> from myapp.forms import ContactForm1, ContactForm2
>>> from myapp.views import ContactWizard
>>> initial = {
...     '0': {'subject': 'Hello', 'sender': 'user@example.com'},
...     '1': {'message': 'Hi there!'}
... }
>>> wiz = ContactWizard.as_view([ContactForm1, ContactForm2], initial_dict=initial)
>>> form1 = wiz.get_form('0')
>>> form2 = wiz.get_form('1')
>>> form1.initial
{'sender': 'user@example.com', 'subject': 'Hello'}
>>> form2.initial
{'message': 'Hi there!'}
```

The `initial_dict` can also take a list of dictionaries for a specific step if the step is a `FormSet`.

Handling files

To handle `FileField` within any step form of the wizard, you have to add a `file_storage` to your `WizardView` subclass.

This storage will temporarily store the uploaded files for the wizard. The `file_storage` attribute should be a `Storage` subclass.

> **Warning:** Please remember to take care of removing old files as the `WizardView` won't remove any files, whether the wizard gets finished correctly or not.

Conditionally view/skip specific steps

`WizardView.condition_dict`

The `as_view()` method accepts a `condition_dict` argument. You can pass a dictionary of boolean values or callables. The key should match the steps names (e.g. '0', '1').

If the value of a specific step is callable it will be called with the `WizardView` instance as the only argument. If the return value is true, the step's form will be used.

This example provides a contact form including a condition. The condition is used to show a message form only if a checkbox in the first step was checked.

The steps are defined in a `forms.py` file:

```
from django import forms

class ContactForm1(forms.Form):
    subject = forms.CharField(max_length=100)
    sender = forms.EmailField()
    leave_message = forms.BooleanField(required=False)

class ContactForm2(forms.Form):
    message = forms.CharField(widget=forms.Textarea)
```

We define our wizard in a `views.py`:

```
from django.shortcuts import render_to_response
from django.contrib.formtools.wizard.views import SessionWizardView

def show_message_form_condition(wizard):
    # try to get the cleaned data of step 1
    cleaned_data = wizard.get_cleaned_data_for_step('0') or {}
    # check if the field ``leave_message`` was checked.
    return cleaned_data.get('leave_message', True)

class ContactWizard(SessionWizardView):

    def done(self, form_list, **kwargs):
        return render_to_response('done.html', {
            'form_data': [form.cleaned_data for form in form_list],
        })
```

We need to add the `ContactWizard` to our `urls.py` file:

```
from django.conf.urls import pattern

from myapp.forms import ContactForm1, ContactForm2
from myapp.views import ContactWizard, show_message_form_condition

contact_forms = [ContactForm1, ContactForm2]
```

```
urlpatterns = patterns('',
    (r'^contact/$', ContactWizard.as_view(contact_forms,
        condition_dict={'1': show_message_form_condition}
    )),
)
```

As you can see, we defined a `show_message_form_condition` next to our `WizardView` subclass and added a `condition_dict` argument to the `as_view()` method. The key refers to the second wizard step (because of the zero based step index).

How to work with ModelForm and ModelFormSet

`WizardView.`**`instance_dict`**

WizardView supports *ModelForms* and *ModelFormSets*. Additionally to `initial_dict`, the `as_view()` method takes an `instance_dict` argument that should contain model instances for steps based on `ModelForm` and querysets for steps based on `ModelFormSet`.

Usage of `NamedUrlWizardView`

class `NamedUrlWizardView`

There is a `WizardView` subclass which adds named-urls support to the wizard. By doing this, you can have single urls for every step.

To use the named urls, you have to change the `urls.py`.

Below you will see an example of a contact wizard with two steps, step 1 with "contactdata" as its name and step 2 with "leavemessage" as its name.

Additionally you have to pass two more arguments to the `as_view()` method:

- `url_name` – the name of the url (as provided in the urls.py)
- `done_step_name` – the name in the url for the done step

Example code for the changed `urls.py` file:

```
from django.conf.urls import url, patterns

from myapp.forms import ContactForm1, ContactForm2
from myapp.views import ContactWizard

named_contact_forms = (
    ('contactdata', ContactForm1),
    ('leavemessage', ContactForm2),
)

contact_wizard = ContactWizard.as_view(named_contact_forms,
    url_name='contact_step', done_step_name='finished')

urlpatterns = patterns('',
    url(r'^contact/(?P<step>.+)/$', contact_wizard, name='contact_step'),
    url(r'^contact/$', contact_wizard, name='contact'),
)
```

Advanced `NamedUrlWizardView` methods

`NamedUrlWizardView.`**`get_step_url`**`(`*`step`*`)`

This method returns the URL for a specific step.

Default implementation:

```
def get_step_url(self, step):
    return reverse(self.url_name, kwargs={'step': step})
```

6.4.9 GeoDjango

GeoDjango intends to be a world-class geographic Web framework. Its goal is to make it as easy as possible to build GIS Web applications and harness the power of spatially enabled data.

GeoDjango Tutorial

Introduction

GeoDjango is an add-on for Django that turns it into a world-class geographic Web framework. GeoDjango strives to make it as simple as possible to create geographic Web applications, like location-based services. Some features include:

- Django model fields for OGC geometries.
- Extensions to Django's ORM for the querying and manipulation of spatial data.
- Loosely-coupled, high-level Python interfaces for GIS geometry operations and data formats.
- Editing of geometry fields inside the admin.

This tutorial assumes a familiarity with Django; thus, if you're brand new to Django please read through the *regular tutorial* to introduce yourself with basic Django concepts.

Note: GeoDjango has special prerequisites overwhat is required by Django – please consult the *installation documentation* for more details.

This tutorial will guide you through the creation of a geographic Web application for viewing the world borders. [1] Some of the code used in this tutorial is taken from and/or inspired by the GeoDjango basic apps project. [2]

Note: Proceed through the tutorial sections sequentially for step-by-step instructions.

Setting Up

Create a Spatial Database

Note: MySQL and Oracle users can skip this section because spatial types are already built into the database.

First, a spatial database needs to be created for our project. If using PostgreSQL and PostGIS, then the following commands will create the database from a *spatial database template*:

[1] Special thanks to Bjørn Sandvik of thematicmapping.org for providing and maintaining this data set.
[2] GeoDjango basic apps was written by Dane Springmeyer, Josh Livni, and Christopher Schmidt.

```
$ createdb -T template_postgis geodjango
```

Note: This command must be issued by a database user that has permissions to create a database. Here is an example set of commands to create such a user:

```
$ sudo su - postgres
$ createuser --createdb geo
$ exit
```

Replace `geo` with the system login user name that will be connecting to the database. For example, `johndoe` if that is the system user that will be running GeoDjango.

Users of SQLite and SpatiaLite should consult the instructions on how to create a *SpatiaLite database*.

Create GeoDjango Project Use the `django-admin.py` script like normal to create a `geodjango` project:

```
$ django-admin.py startproject geodjango
```

With the project initialized, now create a `world` Django application within the `geodjango` project:

```
$ cd geodjango
$ python manage.py startapp world
```

Configure `settings.py` The `geodjango` project settings are stored in the `geodjango/settings.py` file. Edit the database connection settings appropriately:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'geodjango',
        'USER': 'geo',
     }
}
```

In addition, modify the `INSTALLED_APPS` setting to include `django.contrib.admin`, `django.contrib.gis`, and `world` (our newly created application):

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
    'django.contrib.gis',
    'world'
)
```

Geographic Data

World Borders The world borders data is available in this zip file. Create a data directory in the `world` application, download the world borders data, and unzip. On GNU/Linux platforms the following commands should do it:

```
$ mkdir world/data
$ cd world/data
$ wget http://thematicmapping.org/downloads/TM_WORLD_BORDERS-0.3.zip
$ unzip TM_WORLD_BORDERS-0.3.zip
$ cd ../..
```

The world borders ZIP file contains a set of data files collectively known as an ESRI Shapefile, one of the most popular geospatial data formats. When unzipped the world borders data set includes files with the following extensions:

- `.shp`: Holds the vector data for the world borders geometries.
- `.shx`: Spatial index file for geometries stored in the `.shp`.
- `.dbf`: Database file for holding non-geometric attribute data (e.g., integer and character fields).
- `.prj`: Contains the spatial reference information for the geographic data stored in the shapefile.

Use `ogrinfo` to examine spatial data The GDAL `ogrinfo` utility is excellent for examining metadata about shapefiles (or other vector data sources):

```
$ ogrinfo world/data/TM_WORLD_BORDERS-0.3.shp
INFO: Open of 'world/data/TM_WORLD_BORDERS-0.3.shp'
      using driver 'ESRI Shapefile' successful.
1: TM_WORLD_BORDERS-0.3 (Polygon)
```

Here `ogrinfo` is telling us that the shapefile has one layer, and that such layer contains polygon data. To find out more we'll specify the layer name and use the `-so` option to get only important summary information:

```
$ ogrinfo -so world/data/TM_WORLD_BORDERS-0.3.shp TM_WORLD_BORDERS-0.3
INFO: Open of 'world/data/TM_WORLD_BORDERS-0.3.shp'
      using driver 'ESRI Shapefile' successful.

Layer name: TM_WORLD_BORDERS-0.3
Geometry: Polygon
Feature Count: 246
Extent: (-180.000000, -90.000000) - (180.000000, 83.623596)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137.0,298.257223563]],
    PRIMEM["Greenwich",0.0],
    UNIT["Degree",0.0174532925199433]]
FIPS: String (2.0)
ISO2: String (2.0)
ISO3: String (3.0)
UN: Integer (3.0)
NAME: String (50.0)
AREA: Integer (7.0)
POP2005: Integer (10.0)
REGION: Integer (3.0)
SUBREGION: Integer (3.0)
LON: Real (8.3)
LAT: Real (7.3)
```

This detailed summary information tells us the number of features in the layer (246), the geographical extent, the spatial reference system ("SRS WKT"), as well as detailed information for each attribute field. For example, `FIPS: String (2.0)` indicates that there's a `FIPS` character field with a maximum length of 2; similarly, `LON: Real (8.3)` is a floating-point field that holds a maximum of 8 digits up to three decimal places. Although this information

may be found right on the world borders Web site, this shows you how to determine this information yourself when such metadata is not provided.

Geographic Models

Defining a Geographic Model Now that we've examined our world borders data set using `ogrinfo`, we can create a GeoDjango model to represent this data:

```
from django.contrib.gis.db import models

class WorldBorder(models.Model):
    # Regular Django fields corresponding to the attributes in the
    # world borders shapefile.
    name = models.CharField(max_length=50)
    area = models.IntegerField()
    pop2005 = models.IntegerField('Population 2005')
    fips = models.CharField('FIPS Code', max_length=2)
    iso2 = models.CharField('2 Digit ISO', max_length=2)
    iso3 = models.CharField('3 Digit ISO', max_length=3)
    un = models.IntegerField('United Nations Code')
    region = models.IntegerField('Region Code')
    subregion = models.IntegerField('Sub-Region Code')
    lon = models.FloatField()
    lat = models.FloatField()

    # GeoDjango-specific: a geometry field (MultiPolygonField), and
    # overriding the default manager with a GeoManager instance.
    mpoly = models.MultiPolygonField()
    objects = models.GeoManager()

    # Returns the string representation of the model.
    def __unicode__(self):
        return self.name
```

Two important things to note:

1. The `models` module is imported from `django.contrib.gis.db`.
2. The model overrides its default manager with `GeoManager`; this is *required* to perform spatial queries.

When declaring a geometry field on your model the default spatial reference system is WGS84 (meaning the SRID is 4326) – in other words, the field coordinates are in longitude/latitude pairs in units of degrees. If you want the coordinate system to be different, then SRID of the geometry field may be customized by setting the `srid` with an integer corresponding to the coordinate system of your choice.

Run `syncdb` After you've defined your model, it needs to be synced with the spatial database. First, let's look at the SQL that will generate the table for the `WorldBorder` model:

```
$ python manage.py sqlall world
```

This management command should produce the following output:

```
BEGIN;
CREATE TABLE "world_worldborder" (
    "id" serial NOT NULL PRIMARY KEY,
    "name" varchar(50) NOT NULL,
    "area" integer NOT NULL,
    "pop2005" integer NOT NULL,
```

```
    "fips" varchar(2) NOT NULL,
    "iso2" varchar(2) NOT NULL,
    "iso3" varchar(3) NOT NULL,
    "un" integer NOT NULL,
    "region" integer NOT NULL,
    "subregion" integer NOT NULL,
    "lon" double precision NOT NULL,
    "lat" double precision NOT NULL
)
;
SELECT AddGeometryColumn('world_worldborder', 'mpoly', 4326, 'MULTIPOLYGON', 2);
ALTER TABLE "world_worldborder" ALTER "mpoly" SET NOT NULL;
CREATE INDEX "world_worldborder_mpoly_id" ON "world_worldborder" USING GIST ( "mpoly" GIST_GEOMETRY_C
COMMIT;
```

If satisfied, you may then create this table in the database by running the `syncdb` management command:

```
$ python manage.py syncdb
Creating table world_worldborder
Installing custom SQL for world.WorldBorder model
```

The `syncdb` command may also prompt you to create an admin user; go ahead and do so (not required now, may be done at any point in the future using the `createsuperuser` management command).

Importing Spatial Data

This section will show you how to take the data from the world borders shapefile and import it into GeoDjango models using the *LayerMapping data import utility*. There are many different ways to import data in to a spatial database – besides the tools included within GeoDjango, you may also use the following to populate your spatial database:

- ogr2ogr: Command-line utility, included with GDAL, that supports loading a multitude of vector data formats into the PostGIS, MySQL, and Oracle spatial databases.
- shp2pgsql: This utility is included with PostGIS and only supports ESRI shapefiles.

GDAL Interface Earlier we used the `ogrinfo` to explore the contents of the world borders shapefile. Included within GeoDjango is an interface to GDAL's powerful OGR library – in other words, you'll be able explore all the vector data sources that OGR supports via a Pythonic API.

First, invoke the Django shell:

```
$ python manage.py shell
```

If the *World Borders* data was downloaded like earlier in the tutorial, then we can determine the path using Python's built-in `os` module:

```
>>> import os
>>> import world
>>> world_shp = os.path.abspath(os.path.join(os.path.dirname(world.__file__),
...                             'data/TM_WORLD_BORDERS-0.3.shp'))
```

Now, the world borders shapefile may be opened using GeoDjango's `DataSource` interface:

```
>>> from django.contrib.gis.gdal import DataSource
>>> ds = DataSource(world_shp)
>>> print(ds)
/ ... /geodjango/world/data/TM_WORLD_BORDERS-0.3.shp (ESRI Shapefile)
```

Data source objects can have different layers of geospatial features; however, shapefiles are only allowed to have one layer:

```
>>> print(len(ds))
1
>>> lyr = ds[0]
>>> print(lyr)
TM_WORLD_BORDERS-0.3
```

You can see what the geometry type of the layer is and how many features it contains:

```
>>> print(lyr.geom_type)
Polygon
>>> print(len(lyr))
246
```

Note: Unfortunately the shapefile data format does not allow for greater specificity with regards to geometry types. This shapefile, like many others, actually includes `MultiPolygon` geometries in its features. You need to watch out for this when creating your models as a GeoDjango `PolygonField` will not accept a `MultiPolygon` type geometry – thus a `MultiPolygonField` is used in our model's definition instead.

The `Layer` may also have a spatial reference system associated with it – if it does, the `srs` attribute will return a `SpatialReference` object:

```
>>> srs = lyr.srs
>>> print(srs)
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137.0,298.257223563]],
    PRIMEM["Greenwich",0.0],
    UNIT["Degree",0.0174532925199433]]
>>> srs.proj4 # PROJ.4 representation
'+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs '
```

Here we've noticed that the shapefile is in the popular WGS84 spatial reference system – in other words, the data uses units of degrees longitude and latitude.

In addition, shapefiles also support attribute fields that may contain additional data. Here are the fields on the World Borders layer:

```
>>> print(lyr.fields)
['FIPS', 'ISO2', 'ISO3', 'UN', 'NAME', 'AREA', 'POP2005', 'REGION', 'SUBREGION', 'LON', 'LAT']
```

Here we are examining the OGR types (e.g., whether a field is an integer or a string) associated with each of the fields:

```
>>> [fld.__name__ for fld in lyr.field_types]
['OFTString', 'OFTString', 'OFTString', 'OFTInteger', 'OFTString', 'OFTInteger', 'OFTInteger', 'OFTI
```

You can iterate over each feature in the layer and extract information from both the feature's geometry (accessed via the `geom` attribute) as well as the feature's attribute fields (whose **values** are accessed via `get()` method):

```
>>> for feat in lyr:
...     print(feat.get('NAME'), feat.geom.num_points)
...
Guernsey 18
Jersey 26
South Georgia South Sandwich Islands 338
Taiwan 363
```

`Layer` objects may be sliced:

```
>>> lyr[0:2]
[<django.contrib.gis.gdal.feature.Feature object at 0x2f47690>, <django.contrib.gis.gdal.feature.Feat
```

And individual features may be retrieved by their feature ID:

```
>>> feat = lyr[234]
>>> print(feat.get('NAME'))
San Marino
```

Here the boundary geometry for San Marino is extracted and looking exported to WKT and GeoJSON:

```
>>> geom = feat.geom
>>> print(geom.wkt)
POLYGON ((12.415798 43.957954,12.450554 ...
>>> print(geom.json)
{ "type": "Polygon", "coordinates": [ [ [ 12.415798, 43.957954 ], [ 12.450554, 43.979721 ], ...
```

`LayerMapping` We're going to dive right in – create a file called `load.py` inside the `world` application, and insert the following:

```
import os
from django.contrib.gis.utils import LayerMapping
from models import WorldBorder

world_mapping = {
    'fips' : 'FIPS',
    'iso2' : 'ISO2',
    'iso3' : 'ISO3',
    'un' : 'UN',
    'name' : 'NAME',
    'area' : 'AREA',
    'pop2005' : 'POP2005',
    'region' : 'REGION',
    'subregion' : 'SUBREGION',
    'lon' : 'LON',
    'lat' : 'LAT',
    'mpoly' : 'MULTIPOLYGON',
}

world_shp = os.path.abspath(os.path.join(os.path.dirname(__file__), 'data/TM_WORLD_BORDERS-0.3.shp'))

def run(verbose=True):
    lm = LayerMapping(WorldBorder, world_shp, world_mapping,
                      transform=False, encoding='iso-8859-1')

    lm.save(strict=True, verbose=verbose)
```

A few notes about what's going on:

- Each key in the `world_mapping` dictionary corresponds to a field in the `WorldBorder` model, and the value is the name of the shapefile field that data will be loaded from.
- The key `mpoly` for the geometry field is `MULTIPOLYGON`, the geometry type we wish to import as. Even if simple polygons are encountered in the shapefile they will automatically be converted into collections prior to insertion into the database.
- The path to the shapefile is not absolute – in other words, if you move the `world` application (with `data` subdirectory) to a different location, then the script will still work.

- The `transform` keyword is set to `False` because the data in the shapefile does not need to be converted – it's already in WGS84 (SRID=4326).
- The `encoding` keyword is set to the character encoding of string values in the shapefile. This ensures that string values are read and saved correctly from their original encoding system.

Afterwards, invoke the Django shell from the `geodjango` project directory:

```
$ python manage.py shell
```

Next, import the `load` module, call the `run` routine, and watch `LayerMapping` do the work:

```
>>> from world import load
>>> load.run()
```

Try `ogrinspect` Now that you've seen how to define geographic models and import data with the *LayerMapping data import utility*, it's possible to further automate this process with use of the `ogrinspect` management command. The `ogrinspect` command introspects a GDAL-supported vector data source (e.g., a shapefile) and generates a model definition and `LayerMapping` dictionary automatically.

The general usage of the command goes as follows:

```
$ python manage.py ogrinspect [options] <data_source> <model_name> [options]
```

Where `data_source` is the path to the GDAL-supported data source and `model_name` is the name to use for the model. Command-line options may be used to further define how the model is generated.

For example, the following command nearly reproduces the `WorldBorder` model and mapping dictionary created above, automatically:

```
$ python manage.py ogrinspect world/data/TM_WORLD_BORDERS-0.3.shp WorldBorder \
    --srid=4326 --mapping --multi
```

A few notes about the command-line options given above:

- The `--srid=4326` option sets the SRID for the geographic field.
- The `--mapping` option tells `ogrinspect` to also generate a mapping dictionary for use with `LayerMapping`.
- The `--multi` option is specified so that the geographic field is a `MultiPolygonField` instead of just a `PolygonField`.

The command produces the following output, which may be copied directly into the `models.py` of a GeoDjango application:

```
# This is an auto-generated Django model module created by ogrinspect.
from django.contrib.gis.db import models

class WorldBorder(models.Model):
    fips = models.CharField(max_length=2)
    iso2 = models.CharField(max_length=2)
    iso3 = models.CharField(max_length=3)
    un = models.IntegerField()
    name = models.CharField(max_length=50)
    area = models.IntegerField()
    pop2005 = models.IntegerField()
    region = models.IntegerField()
    subregion = models.IntegerField()
    lon = models.FloatField()
    lat = models.FloatField()
```

```
    geom = models.MultiPolygonField(srid=4326)
    objects = models.GeoManager()

# Auto-generated `LayerMapping` dictionary for WorldBorder model
worldborders_mapping = {
    'fips' : 'FIPS',
    'iso2' : 'ISO2',
    'iso3' : 'ISO3',
    'un' : 'UN',
    'name' : 'NAME',
    'area' : 'AREA',
    'pop2005' : 'POP2005',
    'region' : 'REGION',
    'subregion' : 'SUBREGION',
    'lon' : 'LON',
    'lat' : 'LAT',
    'geom' : 'MULTIPOLYGON',
}
```

Spatial Queries

Spatial Lookups GeoDjango extends the Django ORM and allows the use of spatial lookups. Let's do an example where we find the `WorldBorder` model that contains a point. First, fire up the management shell:

```
$ python manage.py shell
```

Now, define a point of interest [3]:

```
>>> pnt_wkt = 'POINT(-95.3385 29.7245)'
```

The `pnt_wkt` string represents the point at -95.3385 degrees longitude, and 29.7245 degrees latitude. The geometry is in a format known as Well Known Text (WKT), an open standard issued by the Open Geospatial Consortium (OGC). [4] Import the `WorldBorder` model, and perform a `contains` lookup using the `pnt_wkt` as the parameter:

```
>>> from world.models import WorldBorder
>>> qs = WorldBorder.objects.filter(mpoly__contains=pnt_wkt)
>>> qs
[<WorldBorder: United States>]
```

Here we retrieved a `GeoQuerySet` that has only one model: the one for the United States (which is what we would expect). Similarly, a *GEOS geometry object* may also be used – here the `intersects` spatial lookup is combined with the `get` method to retrieve only the `WorldBorder` instance for San Marino instead of a queryset:

```
>>> from django.contrib.gis.geos import Point
>>> pnt = Point(12.4604, 43.9420)
>>> sm = WorldBorder.objects.get(mpoly__intersects=pnt)
>>> sm
<WorldBorder: San Marino>
```

The `contains` and `intersects` lookups are just a subset of what's available – the *GeoDjango Database API* documentation has more.

[3] Here the point is for the University of Houston Law Center.
[4] Open Geospatial Consortium, Inc., OpenGIS Simple Feature Specification For SQL.

Automatic Spatial Transformations When querying the spatial database GeoDjango automatically transforms geometries if they're in a different coordinate system. In the following example, the coordinate will be expressed in terms of EPSG SRID 32140, a coordinate system specific to south Texas **only** and in units of **meters** and not degrees:

```
>>> from django.contrib.gis.geos import Point, GEOSGeometry
>>> pnt = Point(954158.1, 4215137.1, srid=32140)
```

Note that `pnt` may also be constructed with EWKT, an "extended" form of WKT that includes the SRID:

```
>>> pnt = GEOSGeometry('SRID=32140;POINT(954158.1 4215137.1)')
```

When using GeoDjango's ORM, it will automatically wrap geometry values in transformation SQL, allowing the developer to work at a higher level of abstraction:

```
>>> qs = WorldBorder.objects.filter(mpoly__intersects=pnt)
>>> print(qs.query) # Generating the SQL
SELECT "world_worldborder"."id", "world_worldborder"."name", "world_worldborder"."area",
"world_worldborder"."pop2005", "world_worldborder"."fips", "world_worldborder"."iso2",
"world_worldborder"."iso3", "world_worldborder"."un", "world_worldborder"."region",
"world_worldborder"."subregion", "world_worldborder"."lon", "world_worldborder"."lat",
"world_worldborder"."mpoly" FROM "world_worldborder"
WHERE ST_Intersects("world_worldborder"."mpoly", ST_Transform(%s, 4326))
>>> qs # printing evaluates the queryset
[<WorldBorder: United States>]
```

Lazy Geometries Geometries come to GeoDjango in a standardized textual representation. Upon access of the geometry field, GeoDjango creates a *GEOS geometry object* <*ref-geos*>, exposing powerful functionality, such as serialization properties for popular geospatial formats:

```
>>> sm = WorldBorder.objects.get(name='San Marino')
>>> sm.mpoly
<MultiPolygon object at 0x24c6798>
>>> sm.mpoly.wkt # WKT
MULTIPOLYGON (((12.4157980000000006 43.9579540000000009, 12.4505540000000003 43.9797209999999978, ...
>>> sm.mpoly.wkb # WKB (as Python binary buffer)
<read-only buffer for 0x1fe2c70, size -1, offset 0 at 0x2564c40>
>>> sm.mpoly.geojson # GeoJSON (requires GDAL)
'{ "type": "MultiPolygon", "coordinates": [ [ [ [ 12.415798, 43.957954 ], [ 12.450554, 43.979721 ], ...
```

This includes access to all of the advanced geometric operations provided by the GEOS library:

```
>>> pnt = Point(12.4604, 43.9420)
>>> sm.mpoly.contains(pnt)
True
>>> pnt.contains(sm.mpoly)
False
```

`GeoQuerySet` Methods

Putting your data on the map

Google

Geographic Admin GeoDjango extends *Django's admin application* to enable support for editing geometry fields.

Basics GeoDjango also supplements the Django admin by allowing users to create and modify geometries on a JavaScript slippy map (powered by OpenLayers).

Let's dive in again – create a file called `admin.py` inside the `world` application, and insert the following:

```
from django.contrib.gis import admin
from models import WorldBorder

admin.site.register(WorldBorder, admin.GeoModelAdmin)
```

Next, edit your `urls.py` in the `geodjango` application folder to look as follows:

```
from django.conf.urls import patterns, url, include
from django.contrib.gis import admin

admin.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
)
```

Start up the Django development server:

```
$ python manage.py runserver
```

Finally, browse to `http://localhost:8000/admin/`, and log in with the admin user created after running `syncdb`. Browse to any of the `WorldBorder` entries – the borders may be edited by clicking on a polygon and dragging the vertexes to the desired position.

`OSMGeoAdmin` With the `OSMGeoAdmin`, GeoDjango uses a Open Street Map layer in the admin. This provides more context (including street and thoroughfare details) than available with the `GeoModelAdmin` (which uses the Vector Map Level 0 WMS data set hosted at OSGeo).

First, there are some important requirements and limitations:

- `OSMGeoAdmin` requires that the *spherical mercator projection be added* to the `spatial_ref_sys` table (PostGIS 1.3 and below, only).
- The PROJ.4 datum shifting files must be installed (see the *PROJ.4 installation instructions* for more details).

If you meet these requirements, then just substitute in the `OSMGeoAdmin` option class in your `admin.py` file:

```
admin.site.register(WorldBorder, admin.OSMGeoAdmin)
```

GeoDjango Installation

Overview

In general, GeoDjango installation requires:

1. *Python and Django*
2. *Spatial database*
3. *Geospatial libraries*

Details for each of the requirements and installation instructions are provided in the sections below. In addition, platform-specific instructions are available for:

- *Mac OS X*

- *Ubuntu & Debian GNU/Linux*
- *Windows*

Use the Source

Because GeoDjango takes advantage of the latest in the open source geospatial software technology, recent versions of the libraries are necessary. If binary packages aren't available for your platform, *installation from source* may be required. When compiling the libraries from source, please follow the directions closely, especially if you're a beginner.

Requirements

Python and Django Because GeoDjango is included with Django, please refer to Django's *installation instructions* for details on how to install.

Spatial database PostgreSQL (with PostGIS), MySQL, Oracle, and SQLite (with SpatiaLite) are the spatial databases currently supported.

Note: PostGIS is recommended, because it is the most mature and feature-rich open source spatial database.

The geospatial libraries required for a GeoDjango installation depends on the spatial database used. The following lists the library requirements, supported versions, and any notes for each of the supported database backends:

Database	Library Requirements	Supported Versions	Notes
PostgreSQL	GEOS, PROJ.4, PostGIS	8.1+	Requires PostGIS.
MySQL	GEOS	5.x	Not OGC-compliant; limited functionality.
Oracle	GEOS	10.2, 11	XE not supported; not tested with 9.
SQLite	GEOS, GDAL, PROJ.4, SpatiaLite	3.6.+	Requires SpatiaLite 2.3+, pysqlite2 2.5+, and Django 1.1.

Geospatial libraries GeoDjango uses and/or provides interfaces for the following open source geospatial libraries:

Program	Description	Required	Supported Versions
GEOS	Geometry Engine Open Source	Yes	3.3, 3.2, 3.1, 3.0
PROJ.4	Cartographic Projections library	Yes (PostgreSQL and SQLite only)	4.7, 4.6, 4.5, 4.4
GDAL	Geospatial Data Abstraction Library	No (but, required for SQLite)	1.9, 1.8, 1.7, 1.6, 1.5
GeoIP	IP-based geolocation library	No	1.4
PostGIS	Spatial extensions for PostgreSQL	Yes (PostgreSQL only)	1.5, 1.4, 1.3
SpatiaLite	Spatial extensions for SQLite	Yes (SQLite only)	3.0, 2.4, 2.3

Install GDAL

While *GDAL* is technically not required, it is *recommended*. Important features of GeoDjango (including the *LayerMapping data import utility*, geometry reprojection, and the geographic admin) depend on its functionality.

Note: The GeoDjango interfaces to GEOS, GDAL, and GeoIP may be used independently of Django. In other words, no database or settings file required – just import them as normal from `django.contrib.gis`.

Building from source

When installing from source on UNIX and GNU/Linux systems, please follow the installation instructions carefully, and install the libraries in the given order. If using MySQL or Oracle as the spatial database, only GEOS is required.

Note: On Linux platforms, it may be necessary to run the `ldconfig` command after installing each library. For example:

```
$ sudo make install
$ sudo ldconfig
```

Note: OS X users are required to install Apple Developer Tools in order to compile software from source. This is typically included on your OS X installation DVDs.

GEOS GEOS is a C++ library for performing geometric operations, and is the default internal geometry representation used by GeoDjango (it's behind the "lazy" geometries). Specifically, the C API library is called (e.g., `libgeos_c.so`) directly from Python using ctypes.

First, download GEOS 3.2 from the refractions Web site and untar the source archive:

```
$ wget http://download.osgeo.org/geos/geos-3.3.0.tar.bz2
$ tar xjf geos-3.3.0.tar.bz2
```

Next, change into the directory where GEOS was unpacked, run the configure script, compile, and install:

```
$ cd geos-3.3.0
$ ./configure
$ make
$ sudo make install
$ cd ..
```

Troubleshooting

Can't find GEOS library When GeoDjango can't find GEOS, this error is raised:

```
ImportError: Could not find the GEOS library (tried "geos_c"). Try setting GEOS_LIBRARY_PATH in your
```

The most common solution is to properly configure your *Library environment settings or* set *GEOS_LIBRARY_PATH* in your settings.

If using a binary package of GEOS (e.g., on Ubuntu), you may need to *Install binutils*.

`GEOS_LIBRARY_PATH` If your GEOS library is in a non-standard location, or you don't want to modify the system's library path then the `GEOS_LIBRARY_PATH` setting may be added to your Django settings file with the full path to the GEOS C library. For example:

```
GEOS_LIBRARY_PATH = '/home/bob/local/lib/libgeos_c.so'
```

Note: The setting must be the *full* path to the **C** shared library; in other words you want to use `libgeos_c.so`, not `libgeos.so`.

PROJ.4 PROJ.4 is a library for converting geospatial data to different coordinate reference systems.

First, download the PROJ.4 source code and datum shifting files [5]:

```
$ wget http://download.osgeo.org/proj/proj-4.7.0.tar.gz
$ wget http://download.osgeo.org/proj/proj-datumgrid-1.5.zip
```

Next, untar the source code archive, and extract the datum shifting files in the `nad` subdirectory. This must be done *prior* to configuration:

```
$ tar xzf proj-4.7.0.tar.gz
$ cd proj-4.7.0/nad
$ unzip ../../proj-datumgrid-1.5.zip
$ cd ..
```

Finally, configure, make and install PROJ.4:

```
$ ./configure
$ make
$ sudo make install
$ cd ..
```

PostGIS PostGIS adds geographic object support to PostgreSQL, turning it into a spatial database. *GEOS* and *PROJ.4* should be installed prior to building PostGIS.

Note: The psycopg2 module is required for use as the database adaptor when using GeoDjango with PostGIS.

First download the source archive, and extract:

```
$ wget http://postgis.refractions.net/download/postgis-1.5.2.tar.gz
$ tar xzf postgis-1.5.2.tar.gz
$ cd postgis-1.5.2
```

Next, configure, make and install PostGIS:

```
$ ./configure
```

Finally, make and install:

```
$ make
$ sudo make install
$ cd ..
```

Note: GeoDjango does not automatically create a spatial database. Please consult the section on *Creating a spatial database template for PostGIS* for more information.

GDAL GDAL is an excellent open source geospatial library that has support for reading most vector and raster spatial data formats. Currently, GeoDjango only supports *GDAL's vector data* capabilities [6]. *GEOS* and *PROJ.4* should be installed prior to building GDAL.

First download the latest GDAL release version and untar the archive:

[5] The datum shifting files are needed for converting data to and from certain projections. For example, the PROJ.4 string for the Google projection (900913 or 3857) requires the `null` grid file only included in the extra datum shifting files. It is easier to install the shifting files now, then to have debug a problem caused by their absence later.

[6] Specifically, GeoDjango provides support for the OGR library, a component of GDAL.

```
$ wget http://download.osgeo.org/gdal/gdal-1.9.1.tar.gz
$ tar xzf gdal-1.9.1.tar.gz
$ cd gdal-1.9.1
```

Configure, make and install:

```
$ ./configure
$ make # Go get some coffee, this takes a while.
$ sudo make install
$ cd ..
```

Note: Because GeoDjango has it's own Python interface, the preceding instructions do not build GDAL's own Python bindings. The bindings may be built by adding the `--with-python` flag when running `configure`. See GDAL/OGR In Python for more information on GDAL's bindings.

If you have any problems, please see the troubleshooting section below for suggestions and solutions.

Troubleshooting

Can't find GDAL library When GeoDjango can't find the GDAL library, the `HAS_GDAL` flag will be false:

```
>>> from django.contrib.gis import gdal
>>> gdal.HAS_GDAL
False
```

The solution is to properly configure your *Library environment settings or* set *GDAL_LIBRARY_PATH* in your settings.

`GDAL_LIBRARY_PATH` If your GDAL library is in a non-standard location, or you don't want to modify the system's library path then the `GDAL_LIBRARY_PATH` setting may be added to your Django settings file with the full path to the GDAL library. For example:

```
GDAL_LIBRARY_PATH = '/home/sue/local/lib/libgdal.so'
```

Can't find GDAL data files (`GDAL_DATA`) When installed from source, GDAL versions 1.5.1 and below have an autoconf bug that places data in the wrong location. [7] This can lead to error messages like this:

```
ERROR 4: Unable to open EPSG support file gcs.csv.
...
OGRException: OGR failure.
```

The solution is to set the `GDAL_DATA` environment variable to the location of the GDAL data files before invoking Python (typically `/usr/local/share`; use `gdal-config --datadir` to find out). For example:

```
$ export GDAL_DATA=`gdal-config --datadir`
$ python manage.py shell
```

If using Apache, you may need to add this environment variable to your configuration file:

```
SetEnv GDAL_DATA /usr/local/share
```

[7] See GDAL ticket #2382.

SpatiaLite

Note: Mac OS X users should follow the instructions in the *KyngChaos packages* section, as it is much easier than building from source.

SpatiaLite adds spatial support to SQLite, turning it into a full-featured spatial database. Because SpatiaLite has special requirements, it typically requires SQLite and pysqlite2 (the Python SQLite DB-API adaptor) to be built from source. *GEOS* and *PROJ.4* should be installed prior to building SpatiaLite.

After installation is complete, don't forget to read the post-installation docs on *Creating a spatial database for SpatiaLite*.

SQLite Typically, SQLite packages are not compiled to include the R*Tree module – thus it must be compiled from source. First download the latest amalgamation source archive from the SQLite download page, and extract:

```
$ wget http://sqlite.org/sqlite-amalgamation-3.6.23.1.tar.gz
$ tar xzf sqlite-amalgamation-3.6.23.1.tar.gz
$ cd sqlite-3.6.23.1
```

Next, run the `configure` script – however the `CFLAGS` environment variable needs to be customized so that SQLite knows to build the R*Tree module:

```
$ CFLAGS="-DSQLITE_ENABLE_RTREE=1" ./configure
$ make
$ sudo make install
$ cd ..
```

Note: If using Ubuntu, installing a newer SQLite from source can be very difficult because it links to the existing `libsqlite3.so` in `/usr/lib` which many other packages depend on. Unfortunately, the best solution at this time is to overwrite the existing library by adding `--prefix=/usr` to the `configure` command.

SpatiaLite library (`libspatialite`) and tools (`spatialite`) After SQLite has been built with the R*Tree module enabled, get the latest SpatiaLite library source and tools bundle from the download page:

```
$ wget http://www.gaia-gis.it/gaia-sins/libspatialite-sources/libspatialite-amalgamation-2.3.1.tar.gz
$ wget http://www.gaia-gis.it/gaia-sins/libspatialite-sources/spatialite-tools-2.3.1.tar.gz
$ tar xzf libspatialite-amalgamation-2.3.1.tar.gz
$ tar xzf spatialite-tools-2.3.1.tar.gz
```

Prior to attempting to build, please read the important notes below to see if customization of the `configure` command is necessary. If not, then run the `configure` script, make, and install for the SpatiaLite library:

```
$ cd libspatialite-amalgamation-2.3.1
$ ./configure # May need to modified, see notes below.
$ make
$ sudo make install
$ cd ..
```

Finally, do the same for the SpatiaLite tools:

```
$ cd spatialite-tools-2.3.1
$ ./configure # May need to modified, see notes below.
$ make
$ sudo make install
$ cd ..
```

Note: If you've installed GEOS and PROJ.4 from binary packages, you will have to specify their paths when running the `configure` scripts for *both* the library and the tools (the configure scripts look, by default, in `/usr/local`). For example, on Debian/Ubuntu distributions that have GEOS and PROJ.4 packages, the command would be:

```
$ ./configure --with-proj-include=/usr/include --with-proj-lib=/usr/lib --with-geos-include=/usr/incl
```

Note: For Mac OS X users building from source, the SpatiaLite library *and* tools need to have their `target` configured:

```
$ ./configure --target=macosx
```

pysqlite2 Because SpatiaLite must be loaded as an external extension, it requires the `enable_load_extension` method, which is only available in versions 2.5+. Thus, download pysqlite2 2.6, and untar:

```
$ wget http://pysqlite.googlecode.com/files/pysqlite-2.6.0.tar.gz
$ tar xzf pysqlite-2.6.0.tar.gz
$ cd pysqlite-2.6.0
```

Next, use a text editor (e.g., `emacs` or `vi`) to edit the `setup.cfg` file to look like the following:

```
[build_ext]
#define=
include_dirs=/usr/local/include
library_dirs=/usr/local/lib
libraries=sqlite3
#define=SQLITE_OMIT_LOAD_EXTENSION
```

Note: The important thing here is to make sure you comment out the `define=SQLITE_OMIT_LOAD_EXTENSION` flag and that the `include_dirs` and `library_dirs` settings are uncommented and set to the appropriate path if the SQLite header files and libraries are not in `/usr/include` and `/usr/lib`, respectively.

After modifying `setup.cfg` appropriately, then run the `setup.py` script to build and install:

```
$ sudo python setup.py install
```

Post-installation

Creating a spatial database template for PostGIS Creating a spatial database with PostGIS is different than normal because additional SQL must be loaded to enable spatial functionality. Because of the steps in this process, it's better to create a database template that can be reused later.

First, you need to be able to execute the commands as a privileged database user. For example, you can use the following to become the `postgres` user:

```
$ sudo su - postgres
```

Note: The location *and* name of the PostGIS SQL files (e.g., from `POSTGIS_SQL_PATH` below) depends on the version of PostGIS. PostGIS versions 1.3 and below use `<pg_sharedir>/contrib/lwpostgis.sql`;

whereas version 1.4 uses `<sharedir>/contrib/postgis.sql` and version 1.5 uses `<sharedir>/contrib/postgis-1.5/postgis.sql`.

To complicate matters, *Ubuntu & Debian GNU/Linux* distributions have their own separate directory naming system that changes each release.

The example below assumes PostGIS 1.5, thus you may need to modify `POSTGIS_SQL_PATH` and the name of the SQL file for the specific version of PostGIS you are using.

Once you're a database super user, then you may execute the following commands to create a PostGIS spatial database template:

```
$ POSTGIS_SQL_PATH=`pg_config --sharedir`/contrib/postgis-1.5
# Creating the template spatial database.
$ createdb -E UTF8 template_postgis
$ createlang -d template_postgis plpgsql # Adding PLPGSQL language support.
# Allows non-superusers the ability to create from this template
$ psql -d postgres -c "UPDATE pg_database SET datistemplate='true' WHERE datname='template_postgis';"
# Loading the PostGIS SQL routines
$ psql -d template_postgis -f $POSTGIS_SQL_PATH/postgis.sql
$ psql -d template_postgis -f $POSTGIS_SQL_PATH/spatial_ref_sys.sql
# Enabling users to alter spatial tables.
$ psql -d template_postgis -c "GRANT ALL ON geometry_columns TO PUBLIC;"
$ psql -d template_postgis -c "GRANT ALL ON geography_columns TO PUBLIC;"
$ psql -d template_postgis -c "GRANT ALL ON spatial_ref_sys TO PUBLIC;"
```

These commands may be placed in a shell script for later use; for convenience the following scripts are available:

PostGIS version	Bash shell script
1.3	`create_template_postgis-1.3.sh`
1.4	`create_template_postgis-1.4.sh`
1.5	`create_template_postgis-1.5.sh`
Debian/Ubuntu	`create_template_postgis-debian.sh`

Afterwards, you may create a spatial database by simply specifying `template_postgis` as the template to use (via the `-T` option):

```
$ createdb -T template_postgis <db name>
```

Note: While the `createdb` command does not require database super-user privileges, it must be executed by a database user that has permissions to create databases. You can create such a user with the following command:

```
$ createuser --createdb <user>
```

Creating a spatial database for SpatiaLite After you've installed SpatiaLite, you'll need to create a number of spatial metadata tables in your database in order to perform spatial queries.

If you're using SpatiaLite 3.0 or newer, use the `spatialite` utility to call the `InitSpatiaMetaData()` function, like this:

```
$ spatialite geodjango.db "SELECT InitSpatialMetaData();"
the SPATIAL_REF_SYS table already contains some row(s)
 InitSpatiaMetaData ()error:"table spatial_ref_sys already exists"
0
```

You can safely ignore the error messages shown. When you've done this, you can skip the rest of this section.

If you're using a version of SpatiaLite older than 3.0, you'll need to download a database-initialization file and execute its SQL queries in your database.

First, get it from the appropriate SpatiaLite Resources page (http://www.gaia-gis.it/spatialite-2.3.1/resources.html for 2.3 or http://www.gaia-gis.it/spatialite-2.4.0/ for 2.4):

```
$ wget http://www.gaia-gis.it/spatialite-2.3.1/init_spatialite-2.3.sql.gz
$ gunzip init_spatialite-2.3.sql.gz
```

Then, use the `spatialite` command to initialize a spatial database:

```
$ spatialite geodjango.db < init_spatialite-2.X.sql
```

Note: The parameter `geodjango.db` is the *filename* of the SQLite database you want to use. Use the same in the `DATABASES "name"` key inside your `settings.py`.

Add `django.contrib.gis` to `INSTALLED_APPS` Like other Django contrib applications, you will *only* need to add `django.contrib.gis` to `INSTALLED_APPS` in your settings. This is the so that `gis` templates can be located – if not done, then features such as the geographic admin or KML sitemaps will not function properly.

Add Google projection to `spatial_ref_sys` table

Note: If you're running PostGIS 1.4 or above, you can skip this step. The entry is already included in the default `spatial_ref_sys` table.

In order to conduct database transformations to the so-called "Google" projection (a spherical mercator projection used by Google Maps), an entry must be added to your spatial database's `spatial_ref_sys` table. Invoke the Django shell from your project and execute the `add_srs_entry` function:

```
$ python manage shell
>>> from django.contrib.gis.utils import add_srs_entry
>>> add_srs_entry(900913)
```

Note: In Django 1.1 the name of this function is `add_postgis_srs`.

This adds an entry for the 900913 SRID to the `spatial_ref_sys` (or equivalent) table, making it possible for the spatial database to transform coordinates in this projection. You only need to execute this command *once* per spatial database.

Troubleshooting

If you can't find the solution to your problem here then participate in the community! You can:

- Join the `#geodjango` IRC channel on FreeNode. Please be patient and polite – while you may not get an immediate response, someone will attempt to answer your question as soon as they see it.
- Ask your question on the GeoDjango mailing list.
- File a ticket on the Django trac if you think there's a bug. Make sure to provide a complete description of the problem, versions used, and specify the component as "GIS".

Library environment settings By far, the most common problem when installing GeoDjango is that the external shared libraries (e.g., for GEOS and GDAL) cannot be located. [8] Typically, the cause of this problem is that the operating system isn't aware of the directory where the libraries built from source were installed.

In general, the library path may be set on a per-user basis by setting an environment variable, or by configuring the library path for the entire system.

`LD_LIBRARY_PATH` environment variable A user may set this environment variable to customize the library paths they want to use. The typical library directory for software built from source is `/usr/local/lib`. Thus, `/usr/local/lib` needs to be included in the `LD_LIBRARY_PATH` variable. For example, the user could place the following in their bash profile:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

Setting system library path On GNU/Linux systems, there is typically a file in `/etc/ld.so.conf`, which may include additional paths from files in another directory, such as `/etc/ld.so.conf.d`. As the root user, add the custom library path (like `/usr/local/lib`) on a new line in `ld.so.conf`. This is *one* example of how to do so:

```
$ sudo echo /usr/local/lib >> /etc/ld.so.conf
$ sudo ldconfig
```

For OpenSolaris users, the system library path may be modified using the `crle` utility. Run `crle` with no options to see the current configuration and use `crle -l` to set with the new library path. Be *very* careful when modifying the system library path:

```
# crle -l $OLD_PATH:/usr/local/lib
```

Install `binutils` GeoDjango uses the `find_library` function (from the `ctypes.util` Python module) to discover libraries. The `find_library` routine uses a program called `objdump` (part of the `binutils` package) to verify a shared library on GNU/Linux systems. Thus, if `binutils` is not installed on your Linux system then Python's ctypes may not be able to find your library even if your library path is set correctly and geospatial libraries were built perfectly.

The `binutils` package may be installed on Debian and Ubuntu systems using the following command:

```
$ sudo apt-get install binutils
```

Similarly, on Red Hat and CentOS systems:

```
$ sudo yum install binutils
```

Platform-specific instructions

Mac OS X Because of the variety of packaging systems available for OS X, users have several different options for installing GeoDjango. These options are:

- *Homebrew*
- *KyngChaos packages*
- *Fink*
- *MacPorts*
- *Building from source*

[8] GeoDjango uses the `find_library()` routine from `ctypes.util` to locate shared libraries.

Note: Currently, the easiest and recommended approach for installing GeoDjango on OS X is to use the KyngChaos packages.

This section also includes instructions for installing an upgraded version of *Python* from packages provided by the Python Software Foundation, however, this is not required.

Python Although OS X comes with Python installed, users can use framework installers (2.6 and 2.7 are available) provided by the Python Software Foundation. An advantage to using the installer is that OS X's Python will remain "pristine" for internal operating system use.

Note: You will need to modify the `PATH` environment variable in your `.profile` file so that the new version of Python is used when `python` is entered at the command-line:

```
export PATH=/Library/Frameworks/Python.framework/Versions/Current/bin:$PATH
```

Homebrew Homebrew provides "recipes" for building binaries and packages from source. It provides recipes for the GeoDjango prerequisites on Macintosh computers running OS X. Because Homebrew still builds the software from source, the Apple Developer Tools are required.

Summary:

```
$ brew install postgresql
$ brew install postgis
$ brew install gdal
$ brew install libgeoip
```

KyngChaos packages William Kyngesburye provides a number of geospatial library binary packages that make it simple to get GeoDjango installed on OS X without compiling them from source. However, the Apple Developer Tools are still necessary for compiling the Python database adapters *psycopg2* (for PostGIS) and *pysqlite2* (for SpatiaLite).

Note: SpatiaLite users should consult the *SpatiaLite* section after installing the packages for additional instructions.

Download the framework packages for:

- UnixImageIO
- PROJ
- GEOS
- SQLite3 (includes the SpatiaLite library)
- GDAL

Install the packages in the order they are listed above, as the GDAL and SQLite packages require the packages listed before them. Afterwards, you can also install the KyngChaos binary packages for PostgreSQL and PostGIS.

After installing the binary packages, you'll want to add the following to your `.profile` to be able to run the package programs from the command-line:

```
export PATH=/Library/Frameworks/UnixImageIO.framework/Programs:$PATH
export PATH=/Library/Frameworks/PROJ.framework/Programs:$PATH
export PATH=/Library/Frameworks/GEOS.framework/Programs:$PATH
```

```
export PATH=/Library/Frameworks/SQLite3.framework/Programs:$PATH
export PATH=/Library/Frameworks/GDAL.framework/Programs:$PATH
export PATH=/usr/local/pgsql/bin:$PATH
```

psycopg2 After you've installed the KyngChaos binaries and modified your `PATH`, as described above, `psycopg2` may be installed using the following command:

```
$ sudo pip install psycopg2
```

Note: If you don't have `pip`, follow the the *installation instructions* to install it.

---

pysqlite2 Follow the *pysqlite2* source install instructions, however, when editing the `setup.cfg` use the following instead:

```
[build_ext]
#define=
include_dirs=/Library/Frameworks/SQLite3.framework/unix/include
library_dirs=/Library/Frameworks/SQLite3.framework/unix/lib
libraries=sqlite3
#define=SQLITE_OMIT_LOAD_EXTENSION
```

SpatiaLite When *Creating a spatial database for SpatiaLite*, the `spatialite` program is required. However, instead of attempting to compile the SpatiaLite tools from source, download the SpatiaLite Binaries for OS X, and install `spatialite` in a location available in your `PATH`. For example:

```
$ curl -O http://www.gaia-gis.it/spatialite/spatialite-tools-osx-x86-2.3.1.tar.gz
$ tar xzf spatialite-tools-osx-x86-2.3.1.tar.gz
$ cd spatialite-tools-osx-x86-2.3.1/bin
$ sudo cp spatialite /Library/Frameworks/SQLite3.framework/Programs
```

Finally, for GeoDjango to be able to find the KyngChaos SpatiaLite library, add the following to your `settings.py`:

```
SPATIALITE_LIBRARY_PATH='/Library/Frameworks/SQLite3.framework/SQLite3'
```

Fink Kurt Schwehr has been gracious enough to create GeoDjango packages for users of the Fink package system. The following packages are available, depending on which version of Python you want to use:

- `django-gis-py26`
- `django-gis-py25`
- `django-gis-py24`

MacPorts MacPorts may be used to install GeoDjango prerequisites on Macintosh computers running OS X. Because MacPorts still builds the software from source, the Apple Developer Tools are required.

Summary:

```
$ sudo port install postgresql83-server
$ sudo port install geos
$ sudo port install proj
$ sudo port install postgis
```

```
$ sudo port install gdal +geos
$ sudo port install libgeoip
```

Note: You will also have to modify the `PATH` in your `.profile` so that the MacPorts programs are accessible from the command-line:

```
export PATH=/opt/local/bin:/opt/local/lib/postgresql83/bin
```

In addition, add the `DYLD_FALLBACK_LIBRARY_PATH` setting so that the libraries can be found by Python:

```
export DYLD_FALLBACK_LIBRARY_PATH=/opt/local/lib:/opt/local/lib/postgresql83
```

Ubuntu & Debian GNU/Linux

Note: The PostGIS SQL files are not placed in the PostgreSQL share directory in the Debian and Ubuntu packages. Instead, they're located in a special directory depending on the release. In this case, use the `create_template_postgis-debian.sh` script

Ubuntu

11.10 In Ubuntu 11.10, PostgreSQL was upgraded to 9.1. The installation commands are:

```
$ sudo apt-get install binutils gdal-bin libproj-dev postgresql-9.1-postgis \
    postgresql-server-dev-9.1 python-psycopg2
```

10.04 through 11.04 In Ubuntu 10.04, PostgreSQL was upgraded to 8.4 and GDAL was upgraded to 1.6. Ubuntu 10.04 uses PostGIS 1.4, while Ubuntu 10.10 uses PostGIS 1.5 (with geography support). The installation commands are:

```
$ sudo apt-get install binutils gdal-bin libproj-dev postgresql-8.4-postgis \
    postgresql-server-dev-8.4 python-psycopg2
```

8.10 Use the synaptic package manager to install the following packages:

```
$ sudo apt-get install binutils gdal-bin postgresql-8.3-postgis \
   postgresql-server-dev-8.3 python-psycopg2
```

That's it! For the curious, the required binary prerequisites packages are:

- `binutils`: for ctypes to find libraries
- `postgresql-8.3`
- `postgresql-server-dev-8.3`: for `pg_config`
- `postgresql-8.3-postgis`: for PostGIS 1.3.3
- `libgeos-3.0.0`, and `libgeos-c1`: for GEOS 3.0.0
- `libgdal1-1.5.0`: for GDAL 1.5.0 library
- `proj`: for PROJ 4.6.0 – but no datum shifting files, see note below
- `python-psycopg2`

Optional packages to consider:

- `libgeoip1`: for *GeoIP* support
- `gdal-bin`: for GDAL command line programs like `ogr2ogr`
- `python-gdal` for GDAL's own Python bindings – includes interfaces for raster manipulation

---

Note: On this version of Ubuntu the `proj` package does not come with the datum shifting files installed, which will cause problems with the geographic admin because the `null` datum grid is not available for transforming geometries to the spherical mercator projection. A solution is to download the datum-shifting files, create the grid file, and install it yourself:

```
$ wget http://download.osgeo.org/proj/proj-datumgrid-1.4.tar.gz
$ mkdir nad
$ cd nad
$ tar xzf ../proj-datumgrid-1.4.tar.gz
$ nad2bin null < null.lla
$ sudo cp null /usr/share/proj
```

Otherwise, the Ubuntu `proj` package is fine for general use as long as you do not plan on doing any database transformation of geometries to the Google projection (900913).

Debian

5.0 (Lenny) This version is comparable to Ubuntu *8.10*, so the command is very similar:

```
$ sudo apt-get install binutils libgdal1-1.5.0 postgresql-8.3 \
    postgresql-8.3-postgis postgresql-server-dev-8.3 \
    python-psycopg2 python-setuptools
```

This assumes that you are using PostgreSQL version 8.3. Else, replace `8.3` in the above command with the appropriate PostgreSQL version.

Note: Please read the note in the Ubuntu *8.10* install documentation about the `proj` package – it also applies here because the package does not include the datum shifting files.

Post-installation notes If the PostgreSQL database cluster was not initiated after installing, then it can be created (and started) with the following command:

```
$ sudo pg_createcluster --start 8.3 main
```

Afterwards, the `/etc/init.d/postgresql-8.3` script should be used to manage the starting and stopping of PostgreSQL.

In addition, the SQL files for PostGIS are placed in a different location on Debian 5.0 . Thus when *Creating a spatial database template for PostGIS* either:

- Create a symbolic link to these files:

  ```
  $ sudo ln -s /usr/share/postgresql-8.3-postgis/{lwpostgis,spatial_ref_sys}.sql \
      /usr/share/postgresql/8.3
  ```

  If not running PostgreSQL 8.3, then replace `8.3` in the command above with the correct version.

- Or use the `create_template_postgis-debian.sh` to create the spatial database.

Windows Proceed through the following sections sequentially in order to install GeoDjango on Windows.

Note: These instructions assume that you are using 32-bit versions of all programs. While 64-bit versions of Python and PostgreSQL 9.0 are available, 64-bit versions of spatial libraries, like GEOS and GDAL, are not yet provided by the *OSGeo4W* installer.

Python First, download the latest Python 2.7 installer from the Python Web site. Next, run the installer and keep the defaults – for example, keep 'Install for all users' checked and the installation path set as `C:\Python27`.

Note: You may already have a version of Python installed in `C:\python` as ESRI products sometimes install a copy there. *You should still install a fresh version of Python 2.7.*

PostgreSQL First, download the latest PostgreSQL 9.0 installer from the EnterpriseDB Web site. After downloading, simply run the installer, follow the on-screen directions, and keep the default options unless you know the consequences of changing them.

Note: The PostgreSQL installer creates both a new Windows user to be the 'postgres service account' and a `postgres` database superuser You will be prompted once to set the password for both accounts – make sure to remember it!

When the installer completes, it will ask to launch the Application Stack Builder (ASB) on exit – keep this checked, as it is necessary to install *PostGIS*.

Note: If installed successfully, the PostgreSQL server will run in the background each time the system as started as a Windows service. A *PostgreSQL 9.0* start menu group will created and contains shortcuts for the ASB as well as the 'SQL Shell', which will launch a `psql` command window.

PostGIS From within the Application Stack Builder (to run outside of the installer, *Start* → *Programs* → *PostgreSQL 9.0*), select *PostgreSQL Database Server 9.0 on port 5432* from the drop down menu. Next, expand the *Categories* → *Spatial Extensions* menu tree and select *PostGIS 1.5 for PostgreSQL 9.0*.

After clicking next, you will be prompted to select your mirror, PostGIS will be downloaded, and the PostGIS installer will begin. Select only the default options during install (e.g., do not uncheck the option to create a default PostGIS database).

Note: You will be prompted to enter your `postgres` database superuser password in the 'Database Connection Information' dialog.

---

psycopg2 The `psycopg2` Python module provides the interface between Python and the PostgreSQL database. Download the latest Windows installer for your version of Python and PostgreSQL and run using the default settings. [9]

[9] The `psycopg2` Windows installers are packaged and maintained by Jason Erickson.

OSGeo4W The OSGeo4W installer makes it simple to install the PROJ.4, GDAL, and GEOS libraries required by GeoDjango. First, download the OSGeo4W installer, and run it. Select *Express Web-GIS Install* and click next. In the 'Select Packages' list, ensure that GDAL is selected; MapServer and Apache are also enabled by default, but are not required by GeoDjango and may be unchecked safely. After clicking next, the packages will be automatically downloaded and installed, after which you may exit the installer.

Modify Windows environment In order to use GeoDjango, you will need to add your Python and OSGeo4W directories to your Windows system `Path`, as well as create `GDAL_DATA` and `PROJ_LIB` environment variables. The following set of commands, executable with `cmd.exe`, will set this up:

```
set OSGEO4W_ROOT=C:\OSGeo4W
set PYTHON_ROOT=C:\Python27
set GDAL_DATA=%OSGEO4W_ROOT%\share\gdal
set PROJ_LIB=%OSGEO4W_ROOT%\share\proj
set PATH=%PATH%;%PYTHON_ROOT%;%OSGEO4W_ROOT%\bin
reg ADD "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" /v Path /t REG_EXPAND_SZ
reg ADD "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" /v GDAL_DATA /t REG_EXPAN
reg ADD "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment" /v PROJ_LIB /t REG_EXPAND
```

For your convenience, these commands are available in the executable batch script, `geodjango_setup.bat`.

Note: Administrator privileges are required to execute these commands. To do this, right-click on `geodjango_setup.bat` and select *Run as administrator*. You need to log out and log back in again for the settings to take effect.

Note: If you customized the Python or OSGeo4W installation directories, then you will need to modify the `OSGEO4W_ROOT` and/or `PYTHON_ROOT` variables accordingly.

Install Django and set up database Finally, *install Django* on your system. You do not need to create a spatial database template, as one named `template_postgis` is created for you when installing PostGIS.

To administer the database, you can either use the pgAdmin III program (*Start* → *PostgreSQL 9.0* → *pgAdmin III*) or the SQL Shell (*Start* → *PostgreSQL 9.0* → *SQL Shell*). For example, to create a `geodjango` spatial database and user, the following may be executed from the SQL Shell as the `postgres` user:

```
postgres# CREATE USER geodjango PASSWORD 'my_passwd';
postgres# CREATE DATABASE geodjango OWNER geodjango TEMPLATE template_postgis ENCODING 'utf8';
```

GeoDjango Model API

This document explores the details of the GeoDjango Model API. Throughout this section, we'll be using the following geographic model of a ZIP code as our example:

```
from django.contrib.gis.db import models

class Zipcode(models.Model):
    code = models.CharField(max_length=5)
    poly = models.PolygonField()
    objects = models.GeoManager()
```

Geometry Field Types

Each of the following geometry field types correspond with the OpenGIS Simple Features specification [10].

GeometryField
class GeometryField

PointField
class PointField

LineStringField
class LineStringField

PolygonField
class PolygonField

MultiPointField
class MultiPointField

MultiLineStringField
class MultiLineStringField

MultiPolygonField
class MultiPolygonField

GeometryCollectionField
class GeometryCollectionField

Geometry Field Options

In addition to the regular *Field options* available for Django model fields, geometry fields have the following additional options. All are optional.

srid
GeometryField.**srid**
Sets the SRID [11] (Spatial Reference System Identity) of the geometry field to the given value. Defaults to 4326 (also known as WGS84, units are in degrees of longitude and latitude).

[10] OpenGIS Consortium, Inc., Simple Feature Specification For SQL.
[11] *See id.* at Ch. 2.3.8, p. 39 (Geometry Values and Spatial Reference Systems).

Selecting an SRID Choosing an appropriate SRID for your model is an important decision that the developer should consider carefully. The SRID is an integer specifier that corresponds to the projection system that will be used to interpret the data in the spatial database. [12] Projection systems give the context to the coordinates that specify a location. Although the details of geodesy are beyond the scope of this documentation, the general problem is that the earth is spherical and representations of the earth (e.g., paper maps, Web maps) are not.

Most people are familiar with using latitude and longitude to reference a location on the earth's surface. However, latitude and longitude are angles, not distances. [13] In other words, while the shortest path between two points on a flat surface is a straight line, the shortest path between two points on a curved surface (such as the earth) is an *arc* of a great circle. [14] Thus, additional computation is required to obtain distances in planar units (e.g., kilometers and miles). Using a geographic coordinate system may introduce complications for the developer later on. For example, PostGIS versions 1.4 and below do not have the capability to perform distance calculations between non-point geometries using geographic coordinate systems, e.g., constructing a query to find all points within 5 miles of a county boundary stored as WGS84. [15]

Portions of the earth's surface may projected onto a two-dimensional, or Cartesian, plane. Projected coordinate systems are especially convenient for region-specific applications, e.g., if you know that your database will only cover geometries in North Kansas, then you may consider using projection system specific to that region. Moreover, projected coordinate systems are defined in Cartesian units (such as meters or feet), easing distance calculations.

Note: If you wish to peform arbitrary distance queries using non-point geometries in WGS84, consider upgrading to PostGIS 1.5. For better performance, enable the `GeometryField.geography` keyword so that *geography database type* is used instead.

Additional Resources:

- spatialreference.org: A Django-powered database of spatial reference systems.
- The State Plane Coordinate System: A Web site covering the various projection systems used in the United States. Much of the U.S. spatial data encountered will be in one of these coordinate systems rather than in a geographic coordinate system such as WGS84.

`spatial_index`
`GeometryField.`**`spatial_index`**
Defaults to `True`. Creates a spatial index for the given geometry field.

Note: This is different from the `db_index` field option because spatial indexes are created in a different manner than regular database indexes. Specifically, spatial indexes are typically created using a variant of the R-Tree, while regular database indexes typically use B-Trees.

`dim`
`GeometryField.`**`dim`**
This option may be used for customizing the coordinate dimension of the geometry field. By default, it is set to 2, for representing two-dimensional geometries. For spatial backends that support it, it may be set to 3 for three-dimensonal support.

[12] Typically, SRID integer corresponds to an EPSG (European Petroleum Survey Group) identifier. However, it may also be associated with custom projections defined in spatial database's spatial reference systems table.

[13] Harvard Graduate School of Design, An Overview of Geodesy and Geographic Referencing Systems. This is an excellent resource for an overview of principles relating to geographic and Cartesian coordinate systems.

[14] Terry A. Slocum, Robert B. McMaster, Fritz C. Kessler, & Hugh H. Howard, *Thematic Cartography and Geographic Visualization* (Prentice Hall, 2nd edition), at Ch. 7.1.3.

[15] This limitation does not apply to PostGIS 1.5. It should be noted that even in previous versions of PostGIS, this isn't impossible using GeoDjango; you could for example, take a known point in a projected coordinate system, buffer it to the appropriate radius, and then perform an intersection operation with the buffer transformed to the geographic coordinate system.

Note: At this time 3D support requires that GEOS 3.1 be installed, and is limited only to the PostGIS spatial backend.

geography

GeometryField.**geography**

If set to `True`, this option will create a database column of type geography, rather than geometry. Please refer to the *geography type* section below for more details.

Note: Geography support is limited only to PostGIS 1.5+, and will force the SRID to be 4326.

Geography Type In PostGIS 1.5, the geography type was introduced – it provides native support for spatial features represented with geographic coordinates (e.g., WGS84 longitude/latitude). [16] Unlike the plane used by a geometry type, the geography type uses a spherical representation of its data. Distance and measurement operations performed on a geography column automatically employ great circle arc calculations and return linear units. In other words, when `ST_Distance` is called on two geographies, a value in meters is returned (as opposed to degrees if called on a geometry column in WGS84).

Because geography calculations involve more mathematics, only a subset of the PostGIS spatial lookups are available for the geography type. Practically, this means that in addition to the *distance lookups* only the following additional *spatial lookups* are available for geography columns:

- `bboverlaps`
- `coveredby`
- `covers`
- `intersects`

For more information, the PostGIS documentation contains a helpful section on determining when to use geography data type over geometry data type.

GeoManager

class GeoManager

In order to conduct geographic queries, each geographic model requires a `GeoManager` model manager. This manager allows for the proper SQL construction for geographic queries; thus, without it, all geographic filters will fail. It should also be noted that `GeoManager` is required even if the model does not have a geographic field itself, e.g., in the case of a `ForeignKey` relation to a model with a geographic field. For example, if we had an `Address` model with a `ForeignKey` to our `Zipcode` model:

```
from django.contrib.gis.db import models
from django.contrib.localflavor.us.models import USStateField

class Address(models.Model):
    num = models.IntegerField()
    street = models.CharField(max_length=100)
    city = models.CharField(max_length=100)
    state = USStateField()
    zipcode = models.ForeignKey(Zipcode)
    objects = models.GeoManager()
```

[16] Please refer to the PostGIS Geography Type documentation for more details.

The geographic manager is needed to do spatial queries on related `Zipcode` objects, for example:

```
qs = Address.objects.filter(zipcode__poly__contains='POINT(-104.590948 38.319914)')
```

GeoDjango Database API

Spatial Backends

GeoDjango currently provides the following spatial database backends:

- `django.contrib.gis.db.backends.postgis`
- `django.contrib.gis.db.backends.mysql`
- `django.contrib.gis.db.backends.oracle`
- `django.contrib.gis.db.backends.spatialite`

MySQL Spatial Limitations MySQL's spatial extensions only support bounding box operations (what MySQL calls minimum bounding rectangles, or MBR). Specifically, MySQL does not conform to the OGC standard:

> Currently, MySQL does not implement these functions [`Contains`, `Crosses`, `Disjoint`, `Intersects`, `Overlaps`, `Touches`, `Within`] according to the specification. Those that are implemented return the same result as the corresponding MBR-based functions.

In other words, while spatial lookups such as `contains` are available in GeoDjango when using MySQL, the results returned are really equivalent to what would be returned when using `bbcontains` on a different spatial backend.

Warning: True spatial indexes (R-trees) are only supported with MyISAM tables on MySQL. [a] In other words, when using MySQL spatial extensions you have to choose between fast spatial lookups and the integrity of your data – MyISAM tables do not support transactions or foreign key constraints.

[a] *See* Creating Spatial Indexes in the MySQL 5.1 Reference Manual:

> For MyISAM tables, `SPATIAL INDEX` creates an R-tree index. For storage engines that support nonspatial indexing of spatial columns, the engine creates a B-tree index. A B-tree index on spatial values will be useful for exact-value lookups, but not for range scans.

Creating and Saving Geographic Models

Here is an example of how to create a geometry object (assuming the `Zipcode` model):

```
>>> from zipcode.models import Zipcode
>>> z = Zipcode(code=77096, poly='POLYGON(( 10 10, 10 20, 20 20, 20 15, 10 10))')
>>> z.save()
```

`GEOSGeometry` objects may also be used to save geometric models:

```
>>> from django.contrib.gis.geos import GEOSGeometry
>>> poly = GEOSGeometry('POLYGON(( 10 10, 10 20, 20 20, 20 15, 10 10))')
>>> z = Zipcode(code=77096, poly=poly)
>>> z.save()
```

Moreover, if the `GEOSGeometry` is in a different coordinate system (has a different SRID value) than that of the field, then it will be implicitly transformed into the SRID of the model's field, using the spatial database's transform procedure:

```
>>> poly_3084 = GEOSGeometry('POLYGON(( 10 10, 10 20, 20 20, 20 15, 10 10))', srid=3084)  # SRID 308
>>> z = Zipcode(code=78212, poly=poly_3084)
>>> z.save()
>>> from django.db import connection
>>> print(connection.queries[-1]['sql']) # printing the last SQL statement executed (requires DEBUG=
INSERT INTO "geoapp_zipcode" ("code", "poly") VALUES (78212, ST_Transform(ST_GeomFromWKB('\\001 ... '
```

Thus, geometry parameters may be passed in using the `GEOSGeometry` object, WKT (Well Known Text [17]), HEX-EWKB (PostGIS specific – a WKB geometry in hexadecimal [18]), and GeoJSON [19] (requires GDAL). Essentially, if the input is not a `GEOSGeometry` object, the geometry field will attempt to create a `GEOSGeometry` instance from the input.

For more information creating `GEOSGeometry` objects, refer to the *GEOS tutorial*.

Spatial Lookups

GeoDjango's lookup types may be used with any manager method like `filter()`, `exclude()`, etc. However, the lookup types unique to GeoDjango are only available on geometry fields. Filters on 'normal' fields (e.g. `CharField`) may be chained with those on geographic fields. Thus, geographic queries take the following general form (assuming the `Zipcode` model used in the *GeoDjango Model API*):

```
>>> qs = Zipcode.objects.filter(<field>__<lookup_type>=<parameter>)
>>> qs = Zipcode.objects.exclude(...)
```

For example:

```
>>> qs = Zipcode.objects.filter(poly__contains=pnt)
```

In this case, `poly` is the geographic field, `contains` is the spatial lookup type, and `pnt` is the parameter (which may be a `GEOSGeometry` object or a string of GeoJSON , WKT, or HEXEWKB).

A complete reference can be found in the *spatial lookup reference*.

Note: GeoDjango constructs spatial SQL with the `GeoQuerySet`, a subclass of `QuerySet`. The `GeoManager` instance attached to your model is what enables use of `GeoQuerySet`.

Distance Queries

Introduction Distance calculations with spatial data is tricky because, unfortunately, the Earth is not flat. Some distance queries with fields in a geographic coordinate system may have to be expressed differently because of limitations in PostGIS. Please see the *Selecting an SRID* section in the *GeoDjango Model API* documentation for more details.

Distance Lookups *Availability*: PostGIS, Oracle, SpatiaLite

The following distance lookups are available:

- `distance_lt`
- `distance_lte`

[17] *See* Open Geospatial Consortium, Inc., OpenGIS Simple Feature Specification For SQL, Document 99-049 (May 5, 1999), at Ch. 3.2.5, p. 3-11 (SQL Textual Representation of Geometry).

[18] *See* PostGIS EWKB, EWKT and Canonical Forms, PostGIS documentation at Ch. 4.1.2.

[19] *See* Howard Butler, Martin Daly, Allan Doyle, Tim Schaub, & Christopher Schmidt, The GeoJSON Format Specification, Revision 1.0 (June 16, 2008).

- distance_gt
- distance_gte
- dwithin

Note: For *measuring*, rather than querying on distances, use the `GeoQuerySet.distance()` method.

Distance lookups take a tuple parameter comprising:

1. A geometry to base calculations from; and
2. A number or `Distance` object containing the distance.

If a `Distance` object is used, it may be expressed in any units (the SQL generated will use units converted to those of the field); otherwise, numeric parameters are assumed to be in the units of the field.

Note: For users of PostGIS 1.4 and below, the routine `ST_Distance_Sphere` is used by default for calculating distances on geographic coordinate systems (e.g., WGS84) – which may only be called with point geometries [20]. Thus, geographic distance lookups on traditional PostGIS geometry columns are only allowed on `PointField` model fields using a point for the geometry parameter.

Note: In PostGIS 1.5, `ST_Distance_Sphere` does *not* limit the geometry types geographic distance queries are performed with. [21] However, these queries may take a long time, as great-circle distances must be calculated on the fly for *every* row in the query. This is because the spatial index on traditional geometry fields cannot be used.

For much better performance on WGS84 distance queries, consider using *geography columns* in your database instead because they are able to use their spatial index in distance queries. You can tell GeoDjango to use a geography column by setting `geography=True` in your field definition.

For example, let's say we have a `SouthTexasCity` model (from the GeoDjango distance tests) on a *projected* coordinate system valid for cities in southern Texas:

```
from django.contrib.gis.db import models

class SouthTexasCity(models.Model):
    name = models.CharField(max_length=30)
    # A projected coordinate system (only valid for South Texas!)
    # is used, units are in meters.
    point = models.PointField(srid=32140)
    objects = models.GeoManager()
```

Then distance queries may be performed as follows:

```
>>> from django.contrib.gis.geos import *
>>> from django.contrib.gis.measure import D # ''D'' is a shortcut for ''Distance''
>>> from geoapp import SouthTexasCity
# Distances will be calculated from this point, which does not have to be projected.
>>> pnt = fromstr('POINT(-96.876369 29.905320)', srid=4326)
# If numeric parameter, units of field (meters in this case) are assumed.
>>> qs = SouthTexasCity.objects.filter(point__distance_lte=(pnt, 7000))
# Find all Cities within 7 km, > 20 miles away, and > 100 chains  away (an obscure unit)
>>> qs = SouthTexasCity.objects.filter(point__distance_lte=(pnt, D(km=7)))
>>> qs = SouthTexasCity.objects.filter(point__distance_gte=(pnt, D(mi=20)))
>>> qs = SouthTexasCity.objects.filter(point__distance_gte=(pnt, D(chain=100)))
```

[20] *See* PostGIS 1.4 documentation on `ST_distance_sphere`.
[21] *See* PostGIS 1.5 documentation on `ST_distance_sphere`.

Compatibility Tables

Spatial Lookups The following table provides a summary of what spatial lookups are available for each spatial database backend.

Lookup Type	PostGIS	Oracle	MySQL [22]	SpatiaLite
bbcontains	X		X	X
bboverlaps	X		X	X
contained	X		X	X
contains	X	X	X	X
contains_properly	X			
coveredby	X	X		
covers	X	X		
crosses	X			X
disjoint	X	X	X	X
distance_gt	X	X		X
distance_gte	X	X		X
distance_lt	X	X		X
distance_lte	X	X		X
dwithin	X	X		
equals	X	X	X	X
exact	X	X	X	X
intersects	X	X	X	X
overlaps	X	X	X	X
relate	X	X		X
same_as	X	X	X	X
touches	X	X	X	X
within	X	X	X	X
left	X			
right	X			
overlaps_left	X			
overlaps_right	X			
overlaps_above	X			
overlaps_below	X			
strictly_above	X			
strictly_below	X			

GeoQuerySet Methods The following table provides a summary of what GeoQuerySet methods are available on each spatial backend. Please note that MySQL does not support any of these methods, and is thus excluded from the table.

Method	PostGIS	Oracle	SpatiaLite
GeoQuerySet.area()	X	X	X
GeoQuerySet.centroid()	X	X	X
GeoQuerySet.collect()	X		
GeoQuerySet.difference()	X	X	X
GeoQuerySet.distance()	X	X	X
GeoQuerySet.envelope()	X		X

Continued on next page

Table 6.2 – continued from previous page

Method	PostGIS	Oracle	SpatiaLite
`GeoQuerySet.extent()`	X	X	
`GeoQuerySet.extent3d()`	X		
`GeoQuerySet.force_rhr()`	X		
`GeoQuerySet.geohash()`	X		
`GeoQuerySet.geojson()`	X		
`GeoQuerySet.gml()`	X	X	X
`GeoQuerySet.intersection()`	X	X	X
`GeoQuerySet.kml()`	X		X
`GeoQuerySet.length()`	X	X	X
`GeoQuerySet.make_line()`	X		
`GeoQuerySet.mem_size()`	X		
`GeoQuerySet.num_geom()`	X	X	X
`GeoQuerySet.num_points()`	X	X	X
`GeoQuerySet.perimeter()`	X	X	
`GeoQuerySet.point_on_surface()`	X	X	X
`GeoQuerySet.reverse_geom()`	X	X	
`GeoQuerySet.scale()`	X		X
`GeoQuerySet.snap_to_grid()`	X		
`GeoQuerySet.svg()`	X		X
`GeoQuerySet.sym_difference()`	X	X	X
`GeoQuerySet.transform()`	X	X	X
`GeoQuerySet.translate()`	X		X
`GeoQuerySet.union()`	X	X	X
`GeoQuerySet.unionagg()`	X	X	X

GeoQuerySet API Reference

class `GeoQuerySet`(*[model=None]*)

Spatial Lookups

Just like when using the *QuerySet API*, interaction with `GeoQuerySet` by *chaining filters*. Instead of the regular Django *Field lookups*, the spatial lookups in this section are available for `GeometryField`.

For an introduction, see the *spatial lookups introduction*. For an overview of what lookups are compatible with a particular spatial backend, refer to the *spatial lookup compatibility table*.

bbcontains *Availability*: PostGIS, MySQL, SpatiaLite

Tests if the geometry field's bounding box completely contains the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__bbcontains=geom)
```

Backend	SQL Equivalent
PostGIS	`poly ~ geom`
MySQL	`MBRContains(poly, geom)`
SpatiaLite	`MbrContains(poly, geom)`

bboverlaps *Availability*: PostGIS, MySQL, SpatiaLite

Tests if the geometry field's bounding box overlaps the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__bboverlaps=geom)
```

Backend	SQL Equivalent
PostGIS	`poly && geom`
MySQL	`MBROverlaps(poly, geom)`
SpatiaLite	`MbrOverlaps(poly, geom)`

contained *Availability*: PostGIS, MySQL, SpatiaLite

Tests if the geometry field's bounding box is completely contained by the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__contained=geom)
```

Backend	SQL Equivalent
PostGIS	`poly @ geom`
MySQL	`MBRWithin(poly, geom)`
SpatiaLite	`MbrWithin(poly, geom)`

contains *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

Tests if the geometry field spatially contains the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__contains=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Contains(poly, geom)`
Oracle	`SDO_CONTAINS(poly, geom)`
MySQL	`MBRContains(poly, geom)`
SpatiaLite	`Contains(poly, geom)`

contains_properly *Availability*: PostGIS

Returns true if the lookup geometry intersects the interior of the geometry field, but not the boundary (or exterior). [24]

Note: Requires PostGIS 1.4 and above.

Example:

```
Zipcode.objects.filter(poly__contains_properly=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_ContainsProperly(poly, geom)`

[24] Refer to the PostGIS `ST_ContainsProperly` documentation for more details.

coveredby *Availability*: PostGIS, Oracle

Tests if no point in the geometry field is outside the lookup geometry. [25]

Example:

```
Zipcode.objects.filter(poly__coveredby=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_CoveredBy(poly, geom)`
Oracle	`SDO_COVEREDBY(poly, geom)`

covers *Availability*: PostGIS, Oracle

Tests if no point in the lookup geometry is outside the geometry field. [3]

Example:

```
Zipcode.objects.filter(poly__covers=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Covers(poly, geom)`
Oracle	`SDO_COVERS(poly, geom)`

crosses *Availability*: PostGIS, SpatiaLite

Tests if the geometry field spatially crosses the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__crosses=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Crosses(poly, geom)`
SpatiaLite	`Crosses(poly, geom)`

disjoint *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

Tests if the geometry field is spatially disjoint from the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__disjoint=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Disjoint(poly, geom)`
Oracle	`SDO_GEOM.RELATE(poly, 'DISJOINT', geom, 0.05)`
MySQL	`MBRDisjoint(poly, geom)`
SpatiaLite	`Disjoint(poly, geom)`

equals *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

exact, same_as *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

[25] For an explanation of this routine, read Quirks of the "Contains" Spatial Predicate by Martin Davis (a PostGIS developer).

intersects *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

Tests if the geometry field spatially intersects the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__intersects=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Intersects(poly, geom)`
Oracle	`SDO_OVERLAPBDYINTERSECT(poly, geom)`
MySQL	`MBRIntersects(poly, geom)`
SpatiaLite	`Intersects(poly, geom)`

overlaps *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

relate *Availability*: PostGIS, Oracle, SpatiaLite

Tests if the geometry field is spatially related to the lookup geometry by the values given in the given pattern. This lookup requires a tuple parameter, `(geom, pattern)`; the form of `pattern` will depend on the spatial backend:

PostGIS & SpatiaLite On these spatial backends the intersection pattern is a string comprising nine characters, which define intersections between the interior, boundary, and exterior of the geometry field and the lookup geometry. The intersection pattern matrix may only use the following characters: `1`, `2`, `T`, `F`, or `*`. This lookup type allows users to "fine tune" a specific geometric relationship consistent with the DE-9IM model. [26]

Example:

```
# A tuple lookup parameter is used to specify the geometry and
# the intersection pattern (the pattern here is for 'contains').
Zipcode.objects.filter(poly__relate(geom, 'T*T***FF*'))
```

PostGIS SQL equivalent:

```
SELECT ... WHERE ST_Relate(poly, geom, 'T*T***FF*')
```

SpatiaLite SQL equivalent:

```
SELECT ... WHERE Relate(poly, geom, 'T*T***FF*')
```

Oracle Here the relation pattern is comprised at least one of the nine relation strings: `TOUCH`, `OVERLAPBDYDISJOINT`, `OVERLAPBDYINTERSECT`, `EQUAL`, `INSIDE`, `COVEREDBY`, `CONTAINS`, `COVERS`, `ON`, and `ANYINTERACT`. Multiple strings may be combined with the logical Boolean operator OR, for example, `'inside+touch'`. [27] The relation strings are case-insensitive.

Example:

```
Zipcode.objects.filter(poly__relate(geom, 'anyinteract'))
```

Oracle SQL equivalent:

```
SELECT ... WHERE SDO_RELATE(poly, geom, 'anyinteract')
```

[26] *See* OpenGIS Simple Feature Specification For SQL, at Ch. 2.1.13.2, p. 2-13 (The Dimensionally Extended Nine-Intersection Model).
[27] *See* SDO_RELATE documentation, from Ch. 11 of the Oracle Spatial User's Guide and Manual.

touches *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

Tests if the geometry field spatially touches the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__touches=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Touches(poly, geom)`
MySQL	`MBRTouches(poly, geom)`
Oracle	`SDO_TOUCH(poly, geom)`
SpatiaLite	`Touches(poly, geom)`

within *Availability*: PostGIS, Oracle, MySQL, SpatiaLite

Tests if the geometry field is spatially within the lookup geometry.

Example:

```
Zipcode.objects.filter(poly__within=geom)
```

Backend	SQL Equivalent
PostGIS	`ST_Within(poly, geom)`
MySQL	`MBRWithin(poly, geom)`
Oracle	`SDO_INSIDE(poly, geom)`
SpatiaLite	`Within(poly, geom)`

left *Availability*: PostGIS

Tests if the geometry field's bounding box is strictly to the left of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__left=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly << geom
```

right *Availability*: PostGIS

Tests if the geometry field's bounding box is strictly to the right of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__right=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly >> geom
```

overlaps_left *Availability*: PostGIS

Tests if the geometry field's bounding box overlaps or is to the left of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_left=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly &< geom
```

overlaps_right *Availability*: PostGIS

Tests if the geometry field's bounding box overlaps or is to the right of the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_right=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly &> geom
```

overlaps_above *Availability*: PostGIS

Tests if the geometry field's bounding box overlaps or is above the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_above=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly |&> geom
```

overlaps_below *Availability*: PostGIS

Tests if the geometry field's bounding box overlaps or is below the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__overlaps_below=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly &<| geom
```

strictly_above *Availability*: PostGIS

Tests if the geometry field's bounding box is strictly above the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__strictly_above=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly |>> geom
```

strictly_below *Availability*: PostGIS

Tests if the geometry field's bounding box is strictly above the lookup geometry's bounding box.

Example:

```
Zipcode.objects.filter(poly__strictly_above=geom)
```

PostGIS equivalent:

```
SELECT ... WHERE poly |>> geom
```

Distance Lookups

Availability: PostGIS, Oracle, SpatiaLite

For an overview on performing distance queries, please refer to the *distance queries introduction*.

Distance lookups take the following form:

```
<field>__<distance lookup>=(<geometry>, <distance value>[, 'spheroid'])
```

The value passed into a distance lookup is a tuple; the first two values are mandatory, and are the geometry to calculate distances to, and a distance value (either a number in units of the field or a `Distance` object). On every distance lookup but `dwithin`, an optional third element, `'spheroid'`, may be included to tell GeoDjango to use the more accurate spheroid distance calculation functions on fields with a geodetic coordinate system (e.g., `ST_Distance_Spheroid` would be used instead of `ST_Distance_Sphere`).

distance_gt Returns models where the distance to the geometry field from the lookup geometry is greater than the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_gt=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance(poly, geom) > 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) > 5`
SpatiaLite	`Distance(poly, geom) > 5`

distance_gte Returns models where the distance to the geometry field from the lookup geometry is greater than or equal to the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_gte=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance(poly, geom) >= 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) >= 5`
SpatiaLite	`Distance(poly, geom) >= 5`

distance_lt Returns models where the distance to the geometry field from the lookup geometry is less than the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_lt=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance(poly, geom) < 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) < 5`
SpatiaLite	`Distance(poly, geom) < 5`

distance_lte Returns models where the distance to the geometry field from the lookup geometry is less than or equal to the given distance value.

Example:

```
Zipcode.objects.filter(poly__distance_lte=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_Distance(poly, geom) <= 5`
Oracle	`SDO_GEOM.SDO_DISTANCE(poly, geom, 0.05) <= 5`
SpatiaLite	`Distance(poly, geom) <= 5`

dwithin Returns models where the distance to the geometry field from the lookup geometry are within the given distance from one another.

Example:

```
Zipcode.objects.filter(poly__dwithin=(geom, D(m=5)))
```

Backend	SQL Equivalent
PostGIS	`ST_DWithin(poly, geom, 5)`
Oracle	`SDO_WITHIN_DISTANCE(poly, geom, 5)`

Note: This lookup is not available on SpatiaLite.

`GeoQuerySet` Methods

`GeoQuerySet` methods specify that a spatial operation be performed on each patial operation on each geographic field in the queryset and store its output in a new attribute on the model (which is generally the name of the `GeoQuerySet` method).

There are also aggregate `GeoQuerySet` methods which return a single value instead of a queryset. This section will describe the API and availability of every `GeoQuerySet` method available in GeoDjango.

Note: What methods are available depend on your spatial backend. See the *compatibility table* for more details.

With a few exceptions, the following keyword arguments may be used with all `GeoQuerySet` methods:

Keyword Argument	Description
field_name	By default, GeoQuerySet methods use the first geographic field encountered in the model. This keyword should be used to specify another geographic field (e.g., field_name='point2') when there are multiple geographic fields in a model. On PostGIS, the field_name keyword may also be used on geometry fields in models that are related via a ForeignKey relation (e.g., field_name='related__point').
model_att	By default, GeoQuerySet methods typically attach their output in an attribute with the same name as the GeoQuerySet method. Setting this keyword with the desired attribute name will override this default behavior. For example, qs = Zipcode.objects.centroid(model_att='c') will attach the centroid of the Zipcode geometry field in a c attribute on every model rather than in a centroid attribute. This keyword is required if a method name clashes with an existing GeoQuerySet method – if you wanted to use the area() method on model with a PolygonField named area, for example.

Measurement *Availability*: PostGIS, Oracle, SpatiaLite

area

GeoQuerySet.**area**(***kwargs*)

Returns the area of the geographic field in an area attribute on each element of this GeoQuerySet.

distance

GeoQuerySet.**distance**(*geom*, ***kwargs*)

This method takes a geometry as a parameter, and attaches a distance attribute to every model in the returned queryset that contains the distance (as a Distance object) to the given geometry.

In the following example (taken from the GeoDjango distance tests), the distance from the Tasmanian city of Hobart to every other PointField in the AustraliaCity queryset is calculated:

```
>>> pnt = AustraliaCity.objects.get(name='Hobart').point
>>> for city in AustraliaCity.objects.distance(pnt): print(city.name, city.distance)
Wollongong 990071.220408 m
Shellharbour 972804.613941 m
Thirroul 1002334.36351 m
Mittagong 975691.632637 m
Batemans Bay 834342.185561 m
Canberra 598140.268959 m
Melbourne 575337.765042 m
Sydney 1056978.87363 m
Hobart 0.0 m
Adelaide 1162031.83522 m
Hillsdale 1049200.46122 m
```

Note: Because the distance attribute is a Distance object, you can easily express the value in the units of your choice. For example, city.distance.mi is the distance value in miles and city.distance.km is the distance value in kilometers. See the *Measurement Objects* for usage details and the list of *Supported units*.

---

length

GeoQuerySet.**length**(***kwargs*)

Returns the length of the geometry field in a length attribute (a Distance object) on each model in the queryset.

perimeter

GeoQuerySet.**perimeter**(***kwargs*)

Returns the perimeter of the geometry field in a `perimeter` attribute (a `Distance` object) on each model in the queryset.

Geometry Relationships The following methods take no arguments, and attach geometry objects each element of the `GeoQuerySet` that is the result of relationship function evaluated on the geometry field.

centroid

GeoQuerySet.**centroid**(***kwargs*)

Availability: PostGIS, Oracle, SpatiaLite

Returns the `centroid` value for the geographic field in a `centroid` attribute on each element of the `GeoQuerySet`.

envelope

GeoQuerySet.**envelope**(***kwargs*)

Availability: PostGIS, SpatiaLite

Returns a geometry representing the bounding box of the geometry field in an `envelope` attribute on each element of the `GeoQuerySet`.

point_on_surface

GeoQuerySet.**point_on_surface**(***kwargs*)

Availability: PostGIS, Oracle, SpatiaLite

Returns a Point geometry guaranteed to lie on the surface of the geometry field in a `point_on_surface` attribute on each element of the queryset; otherwise sets with None.

Geometry Editors

force_rhr

GeoQuerySet.**force_rhr**(***kwargs*)

Availability: PostGIS

Returns a modified version of the polygon/multipolygon in which all of the vertices follow the Right-Hand-Rule, and attaches as a `force_rhr` attribute on each element of the queryset.

reverse_geom

GeoQuerySet.**reverse_geom**(***kwargs*)

Availability: PostGIS, Oracle

Reverse the coordinate order of the geometry field, and attaches as a `reverse` attribute on each element of the queryset.

scale

GeoQuerySet.**scale**(*x*, *y*, *z=0.0*, ***kwargs*)

Availability: PostGIS, SpatiaLite

snap_to_grid

GeoQuerySet.**snap_to_grid**(**args*, ***kwargs*)

Snap all points of the input geometry to the grid. How the geometry is snapped to the grid depends on how many numeric (either float, integer, or long) arguments are given.

Number of Arguments	Description
1	A single size to snap bot the X and Y grids to.
2	X and Y sizes to snap the grid to.
4	X, Y sizes and the corresponding X, Y origins.

transform

GeoQuerySet.**transform**(*srid=4326*, ***kwargs*)

Availability: PostGIS, Oracle, SpatiaLite

The `transform` method transforms the geometry field of a model to the spatial reference system specified by the `srid` parameter. If no `srid` is given, then 4326 (WGS84) is used by default.

Note: Unlike other `GeoQuerySet` methods, `transform` stores its output "in-place". In other words, no new attribute for the transformed geometry is placed on the models.

Note: What spatial reference system an integer SRID corresponds to may depend on the spatial database used. In other words, the SRID numbers used for Oracle are not necessarily the same as those used by PostGIS.

Example:

```
>>> qs = Zipcode.objects.all().transform() # Transforms to WGS84
>>> qs = Zipcode.objects.all().transform(32140) # Transforming to "NAD83 / Texas South Central"
>>> print(qs[0].poly.srid)
32140
>>> print(qs[0].poly)
POLYGON ((234055.1698884720099159 4937796.9232223574072123 ...
```

translate

GeoQuerySet.**translate**(*x*, *y*, *z=0.0*, ***kwargs*)

Availability: PostGIS, SpatiaLite

Translates the geometry field to a new location using the given numeric parameters as offsets.

Geometry Operations *Availability*: PostGIS, Oracle, SpatiaLite

The following methods all take a geometry as a parameter and attach a geometry to each element of the `GeoQuerySet` that is the result of the operation.

difference

GeoQuerySet.**difference**(*geom*)

Returns the spatial difference of the geographic field with the given geometry in a `difference` attribute on each element of the `GeoQuerySet`.

intersection

GeoQuerySet.**intersection**(*geom*)

Returns the spatial intersection of the geographic field with the given geometry in an `intersection` attribute on each element of the `GeoQuerySet`.

sym_difference

GeoQuerySet.**sym_difference**(*geom*)

Returns the symmetric difference of the geographic field with the given geometry in a `sym_difference` attribute on each element of the `GeoQuerySet`.

union

GeoQuerySet.**union**(*geom*)

Returns the union of the geographic field with the given geometry in an `union` attribute on each element of the `GeoQuerySet`.

Geometry Output The following `GeoQuerySet` methods will return an attribute that has the value of the geometry field in each model converted to the requested output format.

geohash

GeoQuerySet.**geohash**(*precision=20*, ***kwargs*)

Attaches a `geohash` attribute to every model the queryset containing the GeoHash representation of the geometry.

geojson

GeoQuerySet.**geojson**(***kwargs*)

Availability: PostGIS

Attaches a `geojson` attribute to every model in the queryset that contains the GeoJSON representation of the geometry.

Keyword Argument	Description
`precision`	It may be used to specify the number of significant digits for the coordinates in the GeoJSON representation – the default value is 8.
`crs`	Set this to `True` if you want the coordinate reference system to be included in the returned GeoJSON.
`bbox`	Set this to `True` if you want the bounding box to be included in the returned GeoJSON.

gml

GeoQuerySet.**gml**(***kwargs*)

Availability: PostGIS, Oracle, SpatiaLite

Attaches a `gml` attribute to every model in the queryset that contains the Geographic Markup Language (GML) representation of the geometry.

Example:

```
>>> qs = Zipcode.objects.all().gml()
>>> print(qs[0].gml)
<gml:Polygon srsName="EPSG:4326"><gml:OuterBoundaryIs>-147.78711,70.245363 ...  -147.78711,70.245363<
```

Keyword Argument	Description
`precision`	This keyword is for PostGIS only. It may be used to specify the number of significant digits for the coordinates in the GML representation – the default value is 8.
`version`	This keyword is for PostGIS only. It may be used to specify the GML version used, and may only be values of 2 or 3. The default value is 2.

kml

GeoQuerySet.**kml**(***kwargs*)
Availability: PostGIS, SpatiaLite

Attaches a `kml` attribute to every model in the queryset that contains the Keyhole Markup Language (KML) representation of the geometry fields. It should be noted that the contents of the KML are transformed to WGS84 if necessary.

Example:

```
>>> qs = Zipcode.objects.all().kml()
>>> print(qs[0].kml)
<Polygon><outerBoundaryIs><LinearRing><coordinates>-103.04135,36.217596,0 ... -103.04135,36.217596,0
```

Keyword Argument	Description
`precision`	This keyword may be used to specify the number of significant digits for the coordinates in the KML representation – the default value is 8.

svg

GeoQuerySet.**svg**(***kwargs*)
Availability: PostGIS, SpatiaLite

Attaches a `svg` attribute to every model in the queryset that contains the Scalable Vector Graphics (SVG) path data of the geometry fields.

Keyword Argument	Description
`relative`	If set to `True`, the path data will be implemented in terms of relative moves. Defaults to `False`, meaning that absolute moves are used instead.
`precision`	This keyword may be used to specify the number of significant digits for the coordinates in the SVG representation – the default value is 8.

Miscellaneous

mem_size

GeoQuerySet.**mem_size**(***kwargs*)
Availability: PostGIS

Returns the memory size (number of bytes) that the geometry field takes in a `mem_size` attribute on each element of the `GeoQuerySet`.

num_geom

GeoQuerySet.**num_geom**(***kwargs*)
Availability: PostGIS, Oracle, SpatiaLite

Returns the number of geometries in a `num_geom` attribute on each element of the `GeoQuerySet` if the geometry field is a collection (e.g., a `GEOMETRYCOLLECTION` or `MULTI*` field); otherwise sets with `None`.

num_points

GeoQuerySet.**num_points**(***kwargs*)
Availability: PostGIS, Oracle, SpatiaLite

Returns the number of points in the first linestring in the geometry field in a `num_points` attribute on each element of the `GeoQuerySet`; otherwise sets with `None`.

Spatial Aggregates

Aggregate Methods

collect

`GeoQuerySet.collect(**kwargs)`
Availability: PostGIS

Returns a `GEOMETRYCOLLECTION` or a `MULTI` geometry object from the geometry column. This is analagous to a simplified version of the `GeoQuerySet.unionagg()` method, except it can be several orders of magnitude faster than peforming a union because it simply rolls up geometries into a collection or multi object, not caring about dissolving boundaries.

extent

`GeoQuerySet.extent(**kwargs)`
Availability: PostGIS, Oracle

Returns the extent of the `GeoQuerySet` as a four-tuple, comprising the lower left coordinate and the upper right coordinate.

Example:

```
>>> qs = City.objects.filter(name__in=('Houston', 'Dallas'))
>>> print(qs.extent())
(-96.8016128540039, 29.7633724212646, -95.3631439208984, 32.782058715820)
```

extent3d

`GeoQuerySet.extent3d(**kwargs)`
Availability: PostGIS

Returns the 3D extent of the `GeoQuerySet` as a six-tuple, comprising the lower left coordinate and upper right coordinate.

Example:

```
>>> qs = City.objects.filter(name__in=('Houston', 'Dallas'))
>>> print(qs.extent3d())
(-96.8016128540039, 29.7633724212646, 0, -95.3631439208984, 32.782058715820, 0)
```

make_line

`GeoQuerySet.make_line(**kwargs)`
Availability: PostGIS

Returns a `LineString` constructed from the point field geometries in the `GeoQuerySet`. Currently, ordering the queryset has no effect.

Example:

```
>>> print(City.objects.filter(name__in=('Houston', 'Dallas')).make_line())
LINESTRING (-95.3631510000000020 29.7633739999999989, -96.8016109999999941 32.7820570000000018)
```

unionagg

`GeoQuerySet.unionagg(**kwargs)`

Availability: PostGIS, Oracle, SpatiaLite

This method returns a `GEOSGeometry` object comprising the union of every geometry in the queryset. Please note that use of `unionagg` is processor intensive and may take a significant amount of time on large querysets.

Note: If the computation time for using this method is too expensive, consider using `GeoQuerySet.collect()` instead.

Example:

```
>>> u = Zipcode.objects.unionagg() # This may take a long time.
>>> u = Zipcode.objects.filter(poly__within=bbox).unionagg() # A more sensible approach.
```

Keyword Argument	Description
`tolerance`	This keyword is for Oracle only. It is for the tolerance value used by the `SDOAGGRTYPE` procedure; the Oracle documentation has more details.

Aggregate Functions Example:

```
>>> from django.contrib.gis.db.models import Extent, Union
>>> WorldBorder.objects.aggregate(Extent('mpoly'), Union('mpoly'))
```

`Collect`
class `Collect`(*geo_field*)
Returns the same as the `GeoQuerySet.collect()` aggregate method.

`Extent`
class `Extent`(*geo_field*)
Returns the same as the `GeoQuerySet.extent()` aggregate method.

`Extent3D`
class `Extent3D`(*geo_field*)
Returns the same as the `GeoQuerySet.extent3d()` aggregate method.

`MakeLine`
class `MakeLine`(*geo_field*)
Returns the same as the `GeoQuerySet.make_line()` aggregate method.

`Union`
class `Union`(*geo_field*)
Returns the same as the `GeoQuerySet.union()` aggregate method.

Measurement Objects

The `django.contrib.gis.measure` module contains objects that allow for convenient representation of distance and area units of measure. [28] Specifically, it implements two objects, `Distance` and `Area` – both of which may be accessed via the `D` and `A` convenience aliases, respectively.

[28] Robert Coup is the initial author of the measure objects, and was inspired by Brian Beck's work in geopy and Geoff Biggs' PhD work on dimensioned units for robotics.

Example

`Distance` objects may be instantiated using a keyword argument indicating the context of the units. In the example below, two different distance objects are instantiated in units of kilometers (`km`) and miles (`mi`):

```
>>> from django.contrib.gis.measure import Distance, D
>>> d1 = Distance(km=5)
>>> print(d1)
5.0 km
>>> d2 = D(mi=5) # 'D' is an alias for 'Distance'
>>> print(d2)
5.0 mi
```

Conversions are easy, just access the preferred unit attribute to get a converted distance quantity:

```
>>> print(d1.mi) # Converting 5 kilometers to miles
3.10685596119
>>> print(d2.km) # Converting 5 miles to kilometers
8.04672
```

Moreover, arithmetic operations may be performed between the distance objects:

```
>>> print(d1 + d2) # Adding 5 miles to 5 kilometers
13.04672 km
>>> print(d2 - d1) # Subtracting 5 kilometers from 5 miles
1.89314403881 mi
```

Two `Distance` objects multiplied together will yield an `Area` object, which uses squared units of measure:

```
>>> a = d1 * d2 # Returns an Area object.
>>> print(a)
40.2336 sq_km
```

To determine what the attribute abbreviation of a unit is, the `unit_attname` class method may be used:

```
>>> print(Distance.unit_attname('US Survey Foot'))
survey_ft
>>> print(Distance.unit_attname('centimeter'))
cm
```

Supported units

Unit Attribute	Full name or alias(es)
km	Kilometre, Kilometer
mi	Mile
m	Meter, Metre
yd	Yard
ft	Foot, Foot (International)
survey_ft	U.S. Foot, US survey foot
inch	Inches
cm	Centimeter
mm	Millimetre, Millimeter
um	Micrometer, Micrometre
british_ft	British foot (Sears 1922)
british_yd	British yard (Sears 1922)

Continued on next page

Table 6.3 – continued from previous page

Unit Attribute	Full name or alias(es)
british_chain_sears	British chain (Sears 1922)
indian_yd	Indian yard, Yard (Indian)
sears_yd	Yard (Sears)
clarke_ft	Clarke's Foot
chain	Chain
chain_benoit	Chain (Benoit)
chain_sears	Chain (Sears)
british_chain_benoit	British chain (Benoit 1895 B)
british_chain_sears_truncated	British chain (Sears 1922 truncated)
gold_coast_ft	Gold Coast foot
link	Link
link_benoit	Link (Benoit)
link_sears	Link (Sears)
clarke_link	Clarke's link
fathom	Fathom
rod	Rod
nm	Nautical Mile
nm_uk	Nautical Mile (UK)
german_m	German legal metre

Note: `Area` attributes are the same as `Distance` attributes, except they are prefixed with `sq_` (area units are square in nature). For example, `Area(sq_m=2)` creates an `Area` object representing two square meters.

Measurement API

`Distance`

class `Distance`(***kwargs*)

To initialize a distance object, pass in a keyword corresponding to the desired *unit attribute name* set with desired value. For example, the following creates a distance object representing 5 miles:

```
>>> dist = Distance(mi=5)
```

`__getattr__`(*unit_att*)

Returns the distance value in units corresponding to the given unit attribute. For example:

```
>>> print(dist.km)
8.04672
```

classmethod `unit_attname`(*unit_name*)

Returns the distance unit attribute name for the given full unit name. For example:

```
>>> Distance.unit_attname('Mile')
'mi'
```

class `D`

Alias for `Distance` class.

`Area`

class Area (***kwargs*)

To initialize a distance object, pass in a keyword corresponding to the desired *unit attribute name* set with desired value. For example, the following creates a distance object representing 5 square miles:

```
>>> a = Area(sq_mi=5)
```

__getattr__(*unit_att*)

Returns the area value in units corresponding to the given unit attribute. For example:

```
>>> print(a.sq_km)
12.949940551680001
```

classmethod unit_attname(*unit_name*)

Returns the area unit attribute name for the given full unit name. For example:

```
>>> Area.unit_attname('Kilometer')
'sq_km'
```

class A

Alias for `Area` class.

GEOS API

Background

What is GEOS? GEOS stands for **G**eometry **E**ngine - **O**pen **S**ource, and is a C++ library, ported from the Java Topology Suite. GEOS implements the OpenGIS Simple Features for SQL spatial predicate functions and spatial operators. GEOS, now an OSGeo project, was initially developed and maintained by Refractions Research of Victoria, Canada.

Features GeoDjango implements a high-level Python wrapper for the GEOS library, its features include:

- A BSD-licensed interface to the GEOS geometry routines, implemented purely in Python using `ctypes`.
- Loosely-coupled to GeoDjango. For example, `GEOSGeometry` objects may be used outside of a django project/application. In other words, no need to have `DJANGO_SETTINGS_MODULE` set or use a database, etc.
- Mutability: `GEOSGeometry` objects may be modified.
- Cross-platform and tested; compatible with Windows, Linux, Solaris, and Mac OS X platforms.

Tutorial

This section contains a brief introduction and tutorial to using `GEOSGeometry` objects.

Creating a Geometry `GEOSGeometry` objects may be created in a few ways. The first is to simply instantiate the object on some spatial input – the following are examples of creating the same geometry from WKT, HEX, WKB, and GeoJSON:

```
>>> from django.contrib.gis.geos import GEOSGeometry
>>> pnt = GEOSGeometry('POINT(5 23)') # WKT
>>> pnt = GEOSGeometry('0101000000000000000000144000000000000003740') # HEX
>>> pnt = GEOSGeometry(buffer('\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x14@\x00\x00\x00\x00\x00
>>> pnt = GEOSGeometry('{ "type": "Point", "coordinates": [ 5.000000, 23.000000 ] }') # GeoJSON
```

Another option is to use the constructor for the specific geometry type that you wish to create. For example, a `Point` object may be created by passing in the X and Y coordinates into its constructor:

```
>>> from django.contrib.gis.geos import Point
>>> pnt = Point(5, 23)
```

Finally, there are `fromstr()` and `fromfile()` factory methods, which return a `GEOSGeometry` object from an input string or a file:

```
>>> from django.contrib.gis.geos import fromstr, fromfile
>>> pnt = fromstr('POINT(5 23)')
>>> pnt = fromfile('/path/to/pnt.wkt')
>>> pnt = fromfile(open('/path/to/pnt.wkt'))
```

Geometries are Pythonic `GEOSGeometry` objects are 'Pythonic', in other words components may be accessed, modified, and iterated over using standard Python conventions. For example, you can iterate over the coordinates in a `Point`:

```
>>> pnt = Point(5, 23)
>>> [coord for coord in pnt]
[5.0, 23.0]
```

With any geometry object, the `GEOSGeometry.coords` property may be used to get the geometry coordinates as a Python tuple:

```
>>> pnt.coords
(5.0, 23.0)
```

You can get/set geometry components using standard Python indexing techniques. However, what is returned depends on the geometry type of the object. For example, indexing on a `LineString` returns a coordinate tuple:

```
>>> from django.contrib.gis.geos import LineString
>>> line = LineString((0, 0), (0, 50), (50, 50), (50, 0), (0, 0))
>>> line[0]
(0.0, 0.0)
>>> line[-2]
(50.0, 0.0)
```

Whereas indexing on a `Polygon` will return the ring (a `LinearRing` object) corresponding to the index:

```
>>> from django.contrib.gis.geos import Polygon
>>> poly = Polygon( ((0.0, 0.0), (0.0, 50.0), (50.0, 50.0), (50.0, 0.0), (0.0, 0.0)) )
>>> poly[0]
<LinearRing object at 0x1044395b0>
>>> poly[0][-2] # second-to-last coordinate of external ring
(50.0, 0.0)
```

In addition, coordinates/components of the geometry may added or modified, just like a Python list:

```
>>> line[0] = (1.0, 1.0)
>>> line.pop()
(0.0, 0.0)
>>> line.append((1.0, 1.0))
>>> line.coords
((1.0, 1.0), (0.0, 50.0), (50.0, 50.0), (50.0, 0.0), (1.0, 1.0))
```

Geometry Objects

GEOSGeometry
class GEOSGeometry (*geo_input*[, *srid=None*])

Parameters

- **geo_input** (*string or buffer*) – Geometry input value
- **srid** (*integer*) – spatial reference identifier

This is the base class for all GEOS geometry objects. It initializes on the given `geo_input` argument, and then assumes the proper geometry subclass (e.g., `GEOSGeometry('POINT(1 1)')` will create a `Point` object).

The following input formats, along with their corresponding Python types, are accepted:

Format	Input Type
WKT / EWKT	`str` or `unicode`
HEX / HEXEWKB	`str` or `unicode`
WKB / EWKB	`buffer`
GeoJSON	`str` or `unicode`

Properties

`GEOSGeometry.`**coords**
Returns the coordinates of the geometry as a tuple.

`GEOSGeometry.`**empty**

Returns whether or not the set of points in the geometry is empty.

`GEOSGeometry.`**geom_type**

Returns a string corresponding to the type of geometry. For example:

```
>>> pnt = GEOSGeometry('POINT(5 23)')
>>> pnt.geom_type
'Point'
```

`GEOSGeometry.`**geom_typeid**

Returns the GEOS geometry type identification number. The following table shows the value for each geometry type:

Geometry	ID
`Point`	0
`LineString`	1
`LinearRing`	2
`Polygon`	3
`MultiPoint`	4
`MultiLineString`	5
`MultiPolygon`	6
`GeometryCollection`	7

`GEOSGeometry.`**num_coords**

Returns the number of coordinates in the geometry.

`GEOSGeometry.`**num_geom**

Returns the number of geometries in this geometry. In other words, will return 1 on anything but geometry collections.

`GEOSGeometry.`**hasz**

Returns a boolean indicating whether the geometry is three-dimensional.

`GEOSGeometry.`**ring**

Returns a boolean indicating whether the geometry is a `LinearRing`.

`GEOSGeometry.`**`simple`**

Returns a boolean indicating whether the geometry is 'simple'. A geometry is simple if and only if it does not intersect itself (except at boundary points). For example, a `LineString` object is not simple if it intersects itself. Thus, `LinearRing` and :class'Polygon' objects are always simple because they do cannot intersect themselves, by definition.

`GEOSGeometry.`**`valid`**

Returns a boolean indicating whether the geometry is valid.

`GEOSGeometry.`**`valid_reason`**

New in version 1.3: *Please see the release notes* Returns a string describing the reason why a geometry is invalid.

`GEOSGeometry.`**`srid`**

Property that may be used to retrieve or set the SRID associated with the geometry. For example:

```
>>> pnt = Point(5, 23)
>>> print(pnt.srid)
None
>>> pnt.srid = 4326
>>> pnt.srid
4326
```

Output Properties The properties in this section export the `GEOSGeometry` object into a different. This output may be in the form of a string, buffer, or even another object.

`GEOSGeometry.`**`ewkt`**

Returns the "extended" Well-Known Text of the geometry. This representation is specific to PostGIS and is a super set of the OGC WKT standard. [29] Essentially the SRID is prepended to the WKT representation, for example `SRID=4326;POINT(5 23)`.

Note: The output from this property does not include the 3dm, 3dz, and 4d information that PostGIS supports in its EWKT representations.

`GEOSGeometry.`**`hex`**

Returns the WKB of this Geometry in hexadecimal form. Please note that the SRID and Z values are not included in this representation because it is not a part of the OGC specification (use the `GEOSGeometry.hexewkb` property instead).

`GEOSGeometry.`**`hexewkb`**

Returns the EWKB of this Geometry in hexadecimal form. This is an extension of the WKB specification that includes SRID and Z values that are a part of this geometry.

Note: GEOS 3.1 is *required* if you want valid 3D HEXEWKB.

`GEOSGeometry.`**`json`**

Returns the GeoJSON representation of the geometry.

Note: Requires GDAL.

[29] *See* PostGIS EWKB, EWKT and Canonical Forms, PostGIS documentation at Ch. 4.1.2.

GEOSGeometry.**geojson**

Alias for GEOSGeometry.json.

GEOSGeometry.**kml**

Returns a KML (Keyhole Markup Language) representation of the geometry. This should only be used for geometries with an SRID of 4326 (WGS84), but this restriction is not enforced.

GEOSGeometry.**ogr**

Returns an OGRGeometry object correspondg to the GEOS geometry.

Note: Requires GDAL.

GEOSGeometry.**wkb**

Returns the WKB (Well-Known Binary) representation of this Geometry as a Python buffer. SRID and Z values are not included, use the GEOSGeometry.ewkb property instead.

GEOSGeometry.**ewkb**

Return the EWKB representation of this Geometry as a Python buffer. This is an extension of the WKB specification that includes any SRID and Z values that are a part of this geometry.

Note: GEOS 3.1 is *required* if you want valid 3D EWKB.

GEOSGeometry.**wkt**

Returns the Well-Known Text of the geometry (an OGC standard).

Spatial Predicate Methods All of the following spatial predicate methods take another GEOSGeometry instance (other) as a parameter, and return a boolean.

GEOSGeometry.**contains**(*other*)

Returns True if GEOSGeometry.within() is False.

GEOSGeometry.**crosses**(*other*)

Returns True if the DE-9IM intersection matrix for the two Geometries is T*T****** (for a point and a curve,a point and an area or a line and an area) 0******** (for two curves).

GEOSGeometry.**disjoint**(*other*)

Returns True if the DE-9IM intersection matrix for the two geometries is FF*FF****.

GEOSGeometry.**equals**(*other*)

Returns True if the DE-9IM intersection matrix for the two geometries is T*F**FFF*.

GEOSGeometry.**equals_exact**(*other*, *tolerance=0*)

Returns true if the two geometries are exactly equal, up to a specified tolerance. The tolerance value should be a floating point number representing the error tolerance in the comparison, e.g., poly1.equals_exact(poly2, 0.001) will compare equality to within one thousandth of a unit.

GEOSGeometry.**intersects**(*other*)

Returns True if GEOSGeometry.disjoint() is False.

GEOSGeometry.**overlaps**(*other*)

Returns true if the DE-9IM intersection matrix for the two geometries is `T*T***T**` (for two points or two surfaces) `1*T***T**` (for two curves).

`GEOSGeometry.`**`relate_pattern`**(*other*, *pattern*)

Returns `True` if the elements in the DE-9IM intersection matrix for this geometry and the other matches the given `pattern` – a string of nine characters from the alphabet: {`T`, `F`, `*`, `0`}.

`GEOSGeometry.`**`touches`**(*other*)

Returns `True` if the DE-9IM intersection matrix for the two geometries is `FT*******`, `F**T*****` or `F***T****`.

`GEOSGeometry.`**`within`**(*other*)

Returns `True` if the DE-9IM intersection matrix for the two geometries is `T*F**F***`.

Topological Methods

`GEOSGeometry.`**`buffer`**(*width*, *quadsegs=8*)

Returns a `GEOSGeometry` that represents all points whose distance from this geometry is less than or equal to the given `width`. The optional `quadsegs` keyword sets the number of segments used to approximate a quarter circle (defaults is 8).

`GEOSGeometry.`**`difference`**(*other*)

Returns a `GEOSGeometry` representing the points making up this geometry that do not make up other.

`GEOSGeometry:intersection(other)`

Returns a `GEOSGeometry` representing the points shared by this geometry and other.

`GEOSGeometry.`**`relate`**(*other*)

Returns the DE-9IM intersection matrix (a string) representing the topological relationship between this geometry and the other.

`GEOSGeometry.`**`simplify`**(*tolerance=0.0*, *preserve_topology=False*)

Returns a new `GEOSGeometry`, simplified using the Douglas-Peucker algorithm to the specified tolerance. A higher tolerance value implies less points in the output. If no tolerance is tolerance provided, it defaults to 0.

By default, this function does not preserve topology - e.g., `Polygon` objects can be split, collapsed into lines or disappear. `Polygon` holes can be created or disappear, and lines can cross. By specifying `preserve_topology=True`, the result will have the same dimension and number of components as the input, however, this is significantly slower.

`GEOSGeometry.`**`sym_difference`**(*other*)

Returns a `GEOSGeometry` combining the points in this geometry not in other, and the points in other not in this geometry.

`GEOSGeometry.`**`union`**(*other*)

Returns a `GEOSGeometry` representing all the points in this geometry and the other.

Topological Properties

`GEOSGeometry.`**`boundary`**

Returns the boundary as a newly allocated Geometry object.

`GEOSGeometry.`**`centroid`**

Returns a `Point` object representing the geometric center of the geometry. The point is not guaranteed to be on the interior of the geometry.

GEOSGeometry.**convex_hull**

Returns the smallest Polygon that contains all the points in the geometry.

GEOSGeometry.**envelope**

Returns a Polygon that represents the bounding envelope of this geometry.

GEOSGeometry.**point_on_surface**

Computes and returns a Point guaranteed to be on the interior of this geometry.

Other Properties & Methods

GEOSGeometry.**area**

This property returns the area of the Geometry.

GEOSGeometry.**extent**

This property returns the extent of this geometry as a 4-tuple, consisting of (xmin, ymin, xmax, ymax).

GEOSGeometry.**clone**()

This method returns a GEOSGeometry that is a clone of the original.

GEOSGeometry.**distance**(*geom*)

Returns the distance between the closest points on this geometry and the given geom (another GEOSGeometry object).

Note: GEOS distance calculations are linear – in other words, GEOS does not perform a spherical calculation even if the SRID specifies a geographic coordinate system.

GEOSGeometry.**length**

Returns the length of this geometry (e.g., 0 for a Point, the length of a LineString, or the circumference of a Polygon).

GEOSGeometry.**prepared**

Note: Support for prepared geometries requires GEOS 3.1.

Returns a GEOS PreparedGeometry for the contents of this geometry. PreparedGeometry objects are optimized for the contains, intersects, and covers operations. Refer to the *Prepared Geometries* documentation for more information.

GEOSGeometry.**srs**

Returns a SpatialReference object corresponding to the SRID of the geometry or None.

Note: Requires GDAL.

GEOSGeometry.**transform**(*ct*, *clone=False*)

Changed in version 1.3: *Please see the release notes* Transforms the geometry according to the given coordinate transformation paramter (ct), which may be an integer SRID, spatial reference WKT string, a PROJ.4 string, a SpatialReference object, or a CoordTransform object. By default, the geometry is transformed in-place and nothing is returned. However if the clone keyword is set, then the geometry is not modified and a transformed clone of the geometry is returned instead.

Note: Requires GDAL.

Note: Prior to 1.3, this method would silently no-op if GDAL was not available. Now, a `GEOSException` is raised as application code relying on this behavior is in error. In addition, use of this method when the SRID is `None` or less than 0 now also generates a `GEOSException`.

Point

class Point (*x*, *y*, *z=None*, *srid=None*)

`Point` objects are instantiated using arguments that represent the component coordinates of the point or with a single sequence coordinates. For example, the following are equivalent:

```
>>> pnt = Point(5, 23)
>>> pnt = Point([5, 23])
```

LineString

class LineString (**args*, ***kwargs*)

`LineString` objects are instantiated using arguments that are either a sequence of coordinates or `Point` objects. For example, the following are equivalent:

```
>>> ls = LineString((0, 0), (1, 1))
>>> ls = LineString(Point(0, 0), Point(1, 1))
```

In addition, `LineString` objects may also be created by passing in a single sequence of coordinate or `Point` objects:

```
>>> ls = LineString( ((0, 0), (1, 1)) )
>>> ls = LineString( [Point(0, 0), Point(1, 1)] )
```

LinearRing

class LinearRing (**args*, ***kwargs*)

`LinearRing` objects are constructed in the exact same way as `LineString` objects, however the coordinates must be *closed*, in other words, the first coordinates must be the same as the last coordinates. For example:

```
>>> ls = LinearRing((0, 0), (0, 1), (1, 1), (0, 0))
```

Notice that `(0, 0)` is the first and last coordinate – if they were not equal, an error would be raised.

Polygon

class Polygon (**args*, ***kwargs*)

`Polygon` objects may be instantiated by passing in one or more parameters that represent the rings of the polygon. The parameters must either be `LinearRing` instances, or a sequence that may be used to construct a `LinearRing`:

```
>>> ext_coords = ((0, 0), (0, 1), (1, 1), (1, 0), (0, 0))
>>> int_coords = ((0.4, 0.4), (0.4, 0.6), (0.6, 0.6), (0.6, 0.4), (0.4, 0.4))
>>> poly = Polygon(ext_coords, int_coords)
>>> poly = Polygon(LinearRing(ext_coords), LinearRing(int_coords))
```

classmethod from_bbox (*bbox*)

Returns a polygon object from the given bounding-box, a 4-tuple comprising (xmin, ymin, xmax, ymax).

num_interior_rings

Returns the number of interior rings in this geometry.

Geometry Collections

MultiPoint

class MultiPoint (**args*, ***kwargs*)

`MultiPoint` objects may be instantiated by passing in one or more `Point` objects as arguments, or a single sequence of `Point` objects:

```
>>> mp = MultiPoint(Point(0, 0), Point(1, 1))
>>> mp = MultiPoint( (Point(0, 0), Point(1, 1)) )
```

MultiLineString

class MultiLineString (**args*, ***kwargs*)

`MultiLineString` objects may be instantiated by passing in one or more `LineString` objects as arguments, or a single sequence of `LineString` objects:

```
>>> ls1 = LineString((0, 0), (1, 1))
>>> ls2 = LineString((2, 2), (3, 3))
>>> mls = MultiLineString(ls1, ls2)
>>> mls = MultiLineString([ls1, ls2])
```

merged

Returns a `LineString` representing the line merge of all the components in this `MultiLineString`.

MultiPolygon

class MultiPolygon (**args*, ***kwargs*)

`MultiPolygon` objects may be instantiated by passing one or more `Polygon` objects as arguments, or a single sequence of `Polygon` objects:

```
>>> p1 = Polygon( ((0, 0), (0, 1), (1, 1), (0, 0)) )
>>> p2 = Polygon( ((1, 1), (1, 2), (2, 2), (1, 1)) )
>>> mp = MultiPolygon(p1, p2)
>>> mp = MultiPolygon([p1, p2])
```

cascaded_union

Returns a `Polygon` that is the union of all of the component polygons in this collection. The algorithm employed is significantly more efficient (faster) than trying to union the geometries together individually. [30]

Note: GEOS 3.1 is *required* to peform cascaded unions.

GeometryCollection

class GeometryCollection (**args*, ***kwargs*)

`GeometryCollection` objects may be instantiated by passing in one or more other `GEOSGeometry` as arguments, or a single sequence of `GEOSGeometry` objects:

```
>>> poly = Polygon( ((0, 0), (0, 1), (1, 1), (0, 0)) )
>>> gc = GeometryCollection(Point(0, 0), MultiPoint(Point(0, 0), Point(1, 1)), poly)
>>> gc = GeometryCollection((Point(0, 0), MultiPoint(Point(0, 0), Point(1, 1)), poly))
```

[30] For more information, read Paul Ramsey's blog post about (Much) Faster Unions in PostGIS 1.4 and Martin Davis' blog post on Fast polygon merging in JTS using Cascaded Union.

Prepared Geometries

In order to obtain a prepared geometry, just access the `GEOSGeometry.prepared` property. Once you have a `PreparedGeometry` instance its spatial predicate methods, listed below, may be used with other `GEOSGeometry` objects. An operation with a prepared geometry can be orders of magnitude faster – the more complex the geometry that is prepared, the larger the speedup in the operation. For more information, please consult the GEOS wiki page on prepared geometries.

Note: GEOS 3.1 is *required* in order to use prepared geometries.

For example:

```
>>> from django.contrib.gis.geos import Point, Polygon
>>> poly = Polygon.from_bbox((0, 0, 5, 5))
>>> prep_poly = poly.prepared
>>> prep_poly.contains(Point(2.5, 2.5))
True
```

PreparedGeometry

class PreparedGeometry

All methods on `PreparedGeometry` take an `other` argument, which must be a `GEOSGeometry` instance.

contains (*other*)

contains_properly (*other*)

covers (*other*)

intersects (*other*)

Geometry Factories

fromfile (*file_h*)

Parameters **file_h** (a Python `file` object or a string path to the file) – input file that contains spatial data

Return type a `GEOSGeometry` corresponding to the spatial data in the file

Example:

```
>>> from django.contrib.gis.geos import fromfile
>>> g = fromfile('/home/bob/geom.wkt')
```

fromstr (*string*[, *srid=None*])

Parameters

- **string** (*string*) – string that contains spatial data
- **srid** (*integer*) – spatial reference identifier

Return type a `GEOSGeometry` corresponding to the spatial data in the string

Example:

```
>>> from django.contrib.gis.geos import fromstr
>>> pnt = fromstr('POINT(-90.5 29.5)', srid=4326)
```

I/O Objects

Reader Objects The reader I/O classes simply return a `GEOSGeometry` instance from the WKB and/or WKT input given to their `read(geom)` method.

class `WKBReader`

Example:

```
>>> from django.contrib.gis.geos import WKBReader
>>> wkb_r = WKBReader()
>>> wkb_r.read('0101000000000000000000F03F000000000000F03F')
<Point object at 0x103a88910>
```

class `WKTReader`

Example:

```
>>> from django.contrib.gis.geos import WKTReader
>>> wkt_r = WKTReader()
>>> wkt_r.read('POINT(1 1)')
<Point object at 0x103a88b50>
```

Writer Objects All writer objects have a `write(geom)` method that returns either the WKB or WKT of the given geometry. In addition, `WKBWriter` objects also have properties that may be used to change the byte order, and or include the SRID and 3D values (in other words, EWKB).

class `WKBWriter`

`WKBWriter` provides the most control over its output. By default it returns OGC-compliant WKB when it's `write` method is called. However, it has properties that allow for the creation of EWKB, a superset of the WKB standard that includes additional information.

`WKBWriter.`**`write`**`(`*geom*`)`

Returns the WKB of the given geometry as a Python `buffer` object. Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> pnt = Point(1, 1)
>>> wkb_w = WKBWriter()
>>> wkb_w.write(pnt)
<read-only buffer for 0x103a898f0, size -1, offset 0 at 0x103a89930>
```

`WKBWriter.`**`write_hex`**`(`*geom*`)`

Returns WKB of the geometry in hexadecimal. Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> pnt = Point(1, 1)
>>> wkb_w = WKBWriter()
>>> wkb_w.write_hex(pnt)
'0101000000000000000000F03F000000000000F03F'
```

`WKBWriter.`**`byteorder`**

This property may be be set to change the byte-order of the geometry representation.

Byteorder Value	Description
0	Big Endian (e.g., compatible with RISC systems)
1	Little Endian (e.g., compatible with x86 systems)

Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> wkb_w = WKBWriter()
>>> pnt = Point(1, 1)
>>> wkb_w.write_hex(pnt)
'0101000000000000000000F03F000000000000F03F'
>>> wkb_w.byteorder = 0
'00000000013FF00000000000003FF0000000000000'
```

`WKBWriter.`**`outdim`**

This property may be set to change the output dimension of the geometry representation. In other words, if you have a 3D geometry then set to 3 so that the Z value is included in the WKB.

Outdim Value	Description
2	The default, output 2D WKB.
3	Output 3D EWKB.

Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> wkb_w = WKBWriter()
>>> wkb_w.outdim
2
>>> pnt = Point(1, 1, 1)
>>> wkb_w.write_hex(pnt) # By default, no Z value included:
'0101000000000000000000F03F000000000000F03F'
>>> wkb_w.outdim = 3 # Tell writer to include Z values
>>> wkb_w.write_hex(pnt)
'01010000800000000000000F03F000000000000F03F000000000000F03F'
```

`WKBWriter.`**`srid`**

Set this property with a boolean to indicate whether the SRID of the geometry should be included with the WKB representation. Example:

```
>>> from django.contrib.gis.geos import Point, WKBWriter
>>> wkb_w = WKBWriter()
>>> pnt = Point(1, 1, srid=4326)
>>> wkb_w.write_hex(pnt) # By default, no SRID included:
'0101000000000000000000F03F000000000000F03F'
>>> wkb_w.srid = True # Tell writer to include SRID
>>> wkb_w.write_hex(pnt)
'0101000020E6100000000000000000F03F000000000000F03F'
```

class `WKTWriter`

`WKTWriter.`**`write`**(*geom*)

Returns the WKT of the given geometry. Example:

```
>>> from django.contrib.gis.geos import Point, WKTWriter
>>> pnt = Point(1, 1)
>>> wkt_w = WKTWriter()
>>> wkt_w.write(pnt)
'POINT (1.0000000000000000 1.0000000000000000)'
```

Settings

GEOS_LIBRARY_PATH A string specifying the location of the GEOS C library. Typically, this setting is only used if the GEOS C library is in a non-standard location (e.g., `/home/bob/lib/libgeos_c.so`).

Note: The setting must be the *full* path to the **C** shared library; in other words you want to use `libgeos_c.so`, not `libgeos.so`.

GDAL API

GDAL stands for **G**eospatial **D**ata **A**bstraction **L**ibrary, and is a veritable "swiss army knife" of GIS data functionality. A subset of GDAL is the OGR Simple Features Library, which specializes in reading and writing vector geographic data in a variety of standard formats.

GeoDjango provides a high-level Python interface for some of the capabilities of OGR, including the reading and coordinate transformation of vector spatial data.

Note: Although the module is named `gdal`, GeoDjango only supports some of the capabilities of OGR. Thus, none of GDAL's features with respect to raster (image) data are supported at this time.

Overview

Sample Data The GDAL/OGR tools described here are designed to help you read in your geospatial data, in order for most of them to be useful you have to have some data to work with. If you're starting out and don't yet have any data of your own to use, GeoDjango comes with a number of simple data sets that you can use for testing. This snippet will determine where these sample files are installed on your computer:

```
>>> import os
>>> import django.contrib.gis
>>> GIS_PATH = os.path.dirname(django.contrib.gis.__file__)
>>> CITIES_PATH = os.path.join(GIS_PATH, 'tests/data/cities/cities.shp')
```

Vector Data Source Objects

`DataSource` `DataSource` is a wrapper for the OGR data source object that supports reading data from a variety of OGR-supported geospatial file formats and data sources using a simple, consistent interface. Each data source is represented by a `DataSource` object which contains one or more layers of data. Each layer, represented by a `Layer` object, contains some number of geographic features (`Feature`), information about the type of features contained in that layer (e.g. points, polygons, etc.), as well as the names and types of any additional fields (`Field`) of data that may be associated with each feature in that layer.

class `DataSource` (*ds_input*)

The constructor for `DataSource` just a single parameter: the path of the file you want to read. However, OGR also supports a variety of more complex data sources, including databases, that may be accessed by passing a special name string instead of a path. For more information, see the OGR Vector Formats documentation. The `name` property of a `DataSource` instance gives the OGR name of the underlying data source that it is using.

Once you've created your `DataSource`, you can find out how many layers of data it contains by accessing the `layer_count` property, or (equivalently) by using the `len()` function. For information on accessing the layers of data themselves, see the next section:

```
>>> from django.contrib.gis.gdal import DataSource
>>> ds = DataSource(CITIES_PATH)
>>> ds.name                           # The exact filename may be different on your computer
'/usr/local/lib/python2.6/site-packages/django/contrib/gis/tests/data/cities/cities.shp'
```

```
>>> ds.layer_count                    # This file only contains one layer
1
```

layer_count

Returns the number of layers in the data source.

name

Returns the name of the data source.

Layer

class Layer

`Layer` is a wrapper for a layer of data in a `DataSource` object. You never create a `Layer` object directly. Instead, you retrieve them from a `DataSource` object, which is essentially a standard Python container of `Layer` objects. For example, you can access a specific layer by its index (e.g. `ds[0]` to access the first layer), or you can iterate over all the layers in the container in a `for` loop. The `Layer` itself acts as a container for geometric features.

Typically, all the features in a given layer have the same geometry type. The `geom_type` property of a layer is an `OGRGeomType` that identifies the feature type. We can use it to print out some basic information about each layer in a `DataSource`:

```
>>> for layer in ds:
...     print('Layer "%s": %i %ss' % (layer.name, len(layer), layer.geom_type.name))
...
Layer "cities": 3 Points
```

The example output is from the cities data source, loaded above, which evidently contains one layer, called `"cities"`, which contains three point features. For simplicity, the examples below assume that you've stored that layer in the variable `layer`:

```
>>> layer = ds[0]
```

name

Returns the name of this layer in the data source.

```
>>> layer.name
'cities'
```

num_feat

Returns the number of features in the layer. Same as `len(layer)`:

```
>>> layer.num_feat
3
```

geom_type

Returns the geometry type of the layer, as an `OGRGeomType` object:

```
>>> layer.geom_type.name
'Point'
```

num_fields

Returns the number of fields in the layer, i.e the number of fields of data associated with each feature in the layer:

```
>>> layer.num_fields
4
```

fields

Returns a list of the names of each of the fields in this layer:

```
>>> layer.fields
['Name', 'Population', 'Density', 'Created']
```

Returns a list of the data types of each of the fields in this layer. These are subclasses of `Field`, discussed below:

```
>>> [ft.__name__ for ft in layer.field_types]
['OFTString', 'OFTReal', 'OFTReal', 'OFTDate']
```

field_widths

Returns a list of the maximum field widths for each of the fields in this layer:

```
>>> layer.field_widths
[80, 11, 24, 10]
```

field_precisions

Returns a list of the numeric precisions for each of the fields in this layer. This is meaningless (and set to zero) for non-numeric fields:

```
>>> layer.field_precisions
[0, 0, 15, 0]
```

extent

Returns the spatial extent of this layer, as an `Envelope` object:

```
>>> layer.extent.tuple
(-104.609252, 29.763374, -95.23506, 38.971823)
```

srs

Property that returns the `SpatialReference` associated with this layer:

```
>>> print(layer.srs)
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.01745329251994295]]
```

If the `Layer` has no spatial reference information associated with it, `None` is returned.

spatial_filter

Property that may be used to retrieve or set a spatial filter for this layer. A spatial filter can only be set with an `OGRGeometry` instance, a 4-tuple extent, or `None`. When set with something other than `None`, only features that intersect the filter will be returned when iterating over the layer:

```
>>> print(layer.spatial_filter)
None
>>> print(len(layer))
3
>>> [feat.get('Name') for feat in layer]
['Pueblo', 'Lawrence', 'Houston']
```

```
>>> ks_extent = (-102.051, 36.99, -94.59, 40.00) # Extent for state of Kansas
>>> layer.spatial_filter = ks_extent
>>> len(layer)
1
>>> [feat.get('Name') for feat in layer]
['Lawrence']
>>> layer.spatial_filter = None
>>> len(layer)
3
```

get_fields()

A method that returns a list of the values of a given field for each feature in the layer:

```
>>> layer.get_fields('Name')
['Pueblo', 'Lawrence', 'Houston']
```

get_geoms([*geos=False*])

A method that returns a list containing the geometry of each feature in the layer. If the optional argument `geos` is set to `True` then the geometries are converted to `GEOSGeometry` objects. Otherwise, they are returned as `OGRGeometry` objects:

```
>>> [pt.tuple for pt in layer.get_geoms()]
[(-104.609252, 38.255001), (-95.23506, 38.971823), (-95.363151, 29.763374)]
```

test_capability(*capability*)

Returns a boolean indicating whether this layer supports the given capability (a string). Examples of valid capability strings include: `'RandomRead'`, `'SequentialWrite'`, `'RandomWrite'`, `'FastSpatialFilter'`, `'FastFeatureCount'`, `'FastGetExtent'`, `'CreateField'`, `'Transactions'`, `'DeleteFeature'`, and `'FastSetNextByIndex'`.

Feature

class Feature

`Feature` wraps an OGR feature. You never create a `Feature` object directly. Instead, you retrieve them from a `Layer` object. Each feature consists of a geometry and a set of fields containing additional properties. The geometry of a field is accessible via its `geom` property, which returns an `OGRGeometry` object. A `Feature` behaves like a standard Python container for its fields, which it returns as `Field` objects: you can access a field directly by its index or name, or you can iterate over a feature's fields, e.g. in a `for` loop.

geom

Returns the geometry for this feature, as an `OGRGeometry` object:

```
>>> city.geom.tuple
(-104.609252, 38.255001)
```

get

A method that returns the value of the given field (specified by name) for this feature, **not** a `Field` wrapper object:

```
>>> city.get('Population')
102121
```

geom_type

Returns the type of geometry for this feature, as an `OGRGeomType` object. This will be the same for all features in a given layer, and is equivalent to the `Layer.geom_type` property of the `Layer` object the feature came from.

num_fields

Returns the number of fields of data associated with the feature. This will be the same for all features in a given layer, and is equivalent to the `Layer.num_fields` property of the `Layer` object the feature came from.

fields

Returns a list of the names of the fields of data associated with the feature. This will be the same for all features in a given layer, and is equivalent to the `Layer.fields` property of the `Layer` object the feature came from.

fid

Returns the feature identifier within the layer:

```
>>> city.fid
0
```

layer_name

Returns the name of the `Layer` that the feature came from. This will be the same for all features in a given layer:

```
>>> city.layer_name
'cities'
```

index

A method that returns the index of the given field name. This will be the same for all features in a given layer:

```
>>> city.index('Population')
1
```

Field
class Field

name

Returns the name of this field:

```
>>> city['Name'].name
'Name'
```

type

Returns the OGR type of this field, as an integer. The `FIELD_CLASSES` dictionary maps these values onto subclasses of `Field`:

```
>>> city['Density'].type
2
```

type_name

Returns a string with the name of the data type of this field:

```
>>> city['Name'].type_name
'String'
```

value

Returns the value of this field. The `Field` class itself returns the value as a string, but each subclass returns the value in the most appropriate form:

```
>>> city['Population'].value
102121
```

width

Returns the width of this field:

```
>>> city['Name'].width
80
```

precision

Returns the numeric precision of this field. This is meaningless (and set to zero) for non-numeric fields:

```
>>> city['Density'].precision
15
```

as_double()

Returns the value of the field as a double (float):

```
>>> city['Density'].as_double()
874.7
```

as_int()

Returns the value of the field as an integer:

```
>>> city['Population'].as_int()
102121
```

as_string()

Returns the value of the field as a string:

```
>>> city['Name'].as_string()
'Pueblo'
```

as_datetime()

Returns the value of the field as a tuple of date and time components:

```
>>> city['Created'].as_datetime()
(c_long(1999), c_long(5), c_long(23), c_long(0), c_long(0), c_long(0), c_long(0))
```

Driver

class Driver(*dr_input*)

The `Driver` class is used internally to wrap an OGR `DataSource` driver.

driver_count

Returns the number of OGR vector drivers currently registered.

OGR Geometries

OGRGeometry `OGRGeometry` objects share similar functionality with `GEOSGeometry` objects, and are thin wrappers around OGR's internal geometry representation. Thus, they allow for more efficient access to data when using `DataSource`. Unlike its GEOS counterpart, `OGRGeometry` supports spatial reference systems and coordinate transformation:

```
>>> from django.contrib.gis.gdal import OGRGeometry
>>> polygon = OGRGeometry('POLYGON((0 0, 5 0, 5 5, 0 5))')
```

class `OGRGeometry`(*geom_input*[, *srs=None*])

This object is a wrapper for the OGR Geometry class. These objects are instantiated directly from the given `geom_input` parameter, which may be a string containing WKT, HEX, GeoJSON, a `buffer` containing WKB data, or an `OGRGeomType` object. These objects are also returned from the `Feature.geom` attribute, when reading vector data from `Layer` (which is in turn a part of a `DataSource`).

classmethod `from_bbox`(*bbox*)

Constructs a `Polygon` from the given bounding-box (a 4-tuple).

`__len__`()

Returns the number of points in a `LineString`, the number of rings in a `Polygon`, or the number of geometries in a `GeometryCollection`. Not applicable to other geometry types.

`__iter__`()

Iterates over the points in a `LineString`, the rings in a `Polygon`, or the geometries in a `GeometryCollection`. Not applicable to other geometry types.

`__getitem__`()

Returns the point at the specified index for a `LineString`, the interior ring at the specified index for a `Polygon`, or the geometry at the specified index in a `GeometryCollection`. Not applicable to other geometry types.

`dimension`

Returns the number of coordinated dimensions of the geometry, i.e. 0 for points, 1 for lines, and so forth:

```
>> polygon.dimension
2
```

`coord_dim`

Returns or sets the coordinate dimension of this geometry. For example, the value would be 2 for two-dimensional geometries.

`geom_count`

Returns the number of elements in this geometry:

```
>>> polygon.geom_count
1
```

`point_count`

Returns the number of points used to describe this geometry:

```
>>> polygon.point_count
4
```

`num_points`

Alias for `point_count`.

`num_coords`

Alias for `point_count`.

`geom_type`

Returns the type of this geometry, as an `OGRGeomType` object.

geom_name

Returns the name of the type of this geometry:

```
>>> polygon.geom_name
'POLYGON'
```

area

Returns the area of this geometry, or 0 for geometries that do not contain an area:

```
>>> polygon.area
25.0
```

envelope

Returns the envelope of this geometry, as an `Envelope` object.

extent

Returns the envelope of this geometry as a 4-tuple, instead of as an `Envelope` object:

```
>>> point.extent
(0.0, 0.0, 5.0, 5.0)
```

srs

This property controls the spatial reference for this geometry, or `None` if no spatial reference system has been assigned to it. If assigned, accessing this property returns a `SpatialReference` object. It may be set with another `SpatialReference` object, or any input that `SpatialReference` accepts. Example:

```
>>> city.geom.srs.name
'GCS_WGS_1984'
```

srid

Returns or sets the spatial reference identifier corresponding to `SpatialReference` of this geometry. Returns `None` if there is no spatial reference information associated with this geometry, or if an SRID cannot be determined.

geos

Returns a `GEOSGeometry` object corresponding to this geometry.

gml

Returns a string representation of this geometry in GML format:

```
>>> OGRGeometry('POINT(1 2)').gml
'<gml:Point><gml:coordinates>1,2</gml:coordinates></gml:Point>'
```

hex

Returns a string representation of this geometry in HEX WKB format:

```
>>> OGRGeometry('POINT(1 2)').hex
'0101000000000000000000F03F0000000000000040'
```

json

Returns a string representation of this geometry in JSON format:

```
>>> OGRGeometry('POINT(1 2)').json
'{ "type": "Point", "coordinates": [ 1.000000, 2.000000 ] }'
```

kml

Returns a string representation of this geometry in KML format.

wkb_size

Returns the size of the WKB buffer needed to hold a WKB representation of this geometry:

```
>>> OGRGeometry('POINT(1 2)').wkb_size
21
```

wkb

Returns a `buffer` containing a WKB representation of this geometry.

wkt

Returns a string representation of this geometry in WKT format.

ewkt

Returns the EWKT representation of this geometry.

clone()

Returns a new `OGRGeometry` clone of this geometry object.

close_rings()

If there are any rings within this geometry that have not been closed, this routine will do so by adding the starting point to the end:

```
>>> triangle = OGRGeometry('LINEARRING (0 0,0 1,1 0)')
>>> triangle.close_rings()
>>> triangle.wkt
'LINEARRING (0 0,0 1,1 0,0 0)'
```

transform(*coord_trans*, *clone=False*)

Transforms this geometry to a different spatial reference system. May take a `CoordTransform` object, a `SpatialReference` object, or any other input accepted by `SpatialReference` (including spatial reference WKT and PROJ.4 strings, or an integer SRID). By default nothing is returned and the geometry is transformed in-place. However, if the *clone* keyword is set to `True` then a transformed clone of this geometry is returned instead.

intersects(*other*)

Returns `True` if this geometry intersects the other, otherwise returns `False`.

equals(*other*)

Returns `True` if this geometry is equivalent to the other, otherwise returns `False`.

disjoint(*other*)

Returns `True` if this geometry is spatially disjoint to (i.e. does not intersect) the other, otherwise returns `False`.

touches(*other*)

Returns `True` if this geometry touches the other, otherwise returns `False`.

crosses(*other*)

Returns `True` if this geometry crosses the other, otherwise returns `False`.

within(*other*)

Returns `True` if this geometry is contained within the other, otherwise returns `False`.

contains(*other*)

Returns `True` if this geometry contains the other, otherwise returns `False`.

overlaps(*other*)

Returns `True` if this geometry overlaps the other, otherwise returns `False`.

boundary()

The boundary of this geometry, as a new `OGRGeometry` object.

convex_hull

The smallest convex polygon that contains this geometry, as a new `OGRGeometry` object.

difference()

Returns the region consisting of the difference of this geometry and the other, as a new `OGRGeometry` object.

intersection()

Returns the region consisting of the intersection of this geometry and the other, as a new `OGRGeometry` object.

sym_difference()

Returns the region consisting of the symmetric difference of this geometry and the other, as a new `OGRGeometry` object.

union()

Returns the region consisting of the union of this geometry and the other, as a new `OGRGeometry` object.

tuple

Returns the coordinates of a point geometry as a tuple, the coordinates of a line geometry as a tuple of tuples, and so forth:

```
>>> OGRGeometry('POINT (1 2)').tuple
(1.0, 2.0)
>>> OGRGeometry('LINESTRING (1 2,3 4)').tuple
((1.0, 2.0), (3.0, 4.0))
```

coords

An alias for `tuple`.

class Point

x

Returns the X coordinate of this point:

```
>>> OGRGeometry('POINT (1 2)').x
1.0
```

y

Returns the Y coordinate of this point:

```
>>> OGRGeometry('POINT (1 2)').y
2.0
```

z

Returns the Z coordinate of this point, or `None` if the the point does not have a Z coordinate:

```
>>> OGRGeometry('POINT (1 2 3)').z
3.0
```

class LineString

x

Returns a list of X coordinates in this line:

```
>>> OGRGeometry('LINESTRING (1 2,3 4)').x
[1.0, 3.0]
```

y

Returns a list of Y coordinates in this line:

```
>>> OGRGeometry('LINESTRING (1 2,3 4)').y
[2.0, 4.0]
```

z

Returns a list of Z coordinates in this line, or `None` if the line does not have Z coordinates:

```
>>> OGRGeometry('LINESTRING (1 2 3,4 5 6)').z
[3.0, 6.0]
```

class Polygon

shell

Returns the shell or exterior ring of this polygon, as a `LinearRing` geometry.

exterior_ring

An alias for `shell`.

centroid

Returns a `Point` representing the centroid of this polygon.

class GeometryCollection

add(*geom*)

Adds a geometry to this geometry collection. Not applicable to other geometry types.

OGRGeomType

class OGRGeomType(*type_input*)

This class allows for the representation of an OGR geometry type in any of several ways:

```
>>> from django.contrib.gis.gdal import OGRGeomType
>>> gt1 = OGRGeomType(3)             # Using an integer for the type
>>> gt2 = OGRGeomType('Polygon')     # Using a string
>>> gt3 = OGRGeomType('POLYGON')     # It's case-insensitive
>>> print(gt1 == 3, gt1 == 'Polygon') # Equivalence works w/non-OGRGeomType objects
True True
```

name

Returns a short-hand string form of the OGR Geometry type:

```
>>> gt1.name
'Polygon'
```

num

Returns the number corresponding to the OGR geometry type:

```
>>> gt1.num
3
```

django

Returns the Django field type (a subclass of GeometryField) to use for storing this OGR type, or `None` if there is no appropriate Django type:

```
>>> gt1.django
'PolygonField'
```

Envelope

class Envelope(**args*)

Represents an OGR Envelope structure that contains the minimum and maximum X, Y coordinates for a rectangle bounding box. The naming of the variables is compatible with the OGR Envelope C structure.

min_x

The value of the minimum X coordinate.

min_y

The value of the maximum X coordinate.

max_x

The value of the minimum Y coordinate.

max_y

The value of the maximum Y coordinate.

ur

The upper-right coordinate, as a tuple.

ll

The lower-left coordinate, as a tuple.

tuple

A tuple representing the envelope.

wkt

A string representing this envelope as a polygon in WKT format.

expand_to_include(*self*, **args*)

Coordinate System Objects

SpatialReference

class SpatialReference(*srs_input*)

Spatial reference objects are initialized on the given `srs_input`, which may be one of the following:

- OGC Well Known Text (WKT) (a string)
- EPSG code (integer or string)
- PROJ.4 string
- A shorthand string for well-known standards (`'WGS84'`, `'WGS72'`, `'NAD27'`, `'NAD83'`)

Example:

```
>>> wgs84 = SpatialReference('WGS84') # shorthand string
>>> wgs84 = SpatialReference(4326) # EPSG code
>>> wgs84 = SpatialReference('EPSG:4326') # EPSG string
>>> proj4 = '+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs '
>>> wgs84 = SpatialReference(proj4) # PROJ.4 string
>>> wgs84 = SpatialReference("""GEOGCS["WGS 84",
DATUM["WGS_1984",
     SPHEROID["WGS 84",6378137,298.257223563,
         AUTHORITY["EPSG","7030"]],
     AUTHORITY["EPSG","6326"]],
 PRIMEM["Greenwich",0,
     AUTHORITY["EPSG","8901"]],
 UNIT["degree",0.01745329251994328,
     AUTHORITY["EPSG","9122"]],
 AUTHORITY["EPSG","4326"]]""") # OGC WKT
```

__getitem__(*target*)

Returns the value of the given string attribute node, `None` if the node doesn't exist. Can also take a tuple as a parameter, (target, child), where child is the index of the attribute in the WKT. For example:

```
>>> wkt = 'GEOGCS["WGS 84", DATUM["WGS_1984, ... AUTHORITY["EPSG","4326"]]')
>>> srs = SpatialReference(wkt) # could also use 'WGS84', or 4326
>>> print(srs['GEOGCS'])
WGS 84
>>> print(srs['DATUM'])
WGS_1984
>>> print(srs['AUTHORITY'])
EPSG
>>> print(srs['AUTHORITY', 1]) # The authority value
4326
>>> print(srs['TOWGS84', 4]) # the fourth value in this wkt
0
>>> print(srs['UNIT|AUTHORITY']) # For the units authority, have to use the pipe symbol.
EPSG
>>> print(srs['UNIT|AUTHORITY', 1]) # The authority value for the units
9122
```

attr_value(*target*, *index=0*)

The attribute value for the given target node (e.g. `'PROJCS'`). The index keyword specifies an index of the child node to return.

auth_name(*target*)

Returns the authority name for the given string target node.

auth_code(*target*)

Returns the authority code for the given string target node.

clone()

Returns a clone of this spatial reference object.

identify_epsg()

This method inspects the WKT of this SpatialReference, and will add EPSG authority nodes where an EPSG identifier is applicable.

from_esri()

Morphs this SpatialReference from ESRI's format to EPSG

to_esri()

Morphs this SpatialReference to ESRI's format.

validate()

Checks to see if the given spatial reference is valid, if not an exception will be raised.

import_epsg(*epsg*)

Import spatial reference from EPSG code.

import_proj(*proj*)

Import spatial reference from PROJ.4 string.

import_user_input(*user_input*)

import_wkt(*wkt*)

Import spatial reference from WKT.

import_xml(*xml*)

Import spatial reference from XML.

name

Returns the name of this Spatial Reference.

srid

Returns the SRID of top-level authority, or `None` if undefined.

linear_name

Returns the name of the linear units.

linear_units

Returns the value of the linear units.

angular_name

Returns the name of the angular units."

angular_units

Returns the value of the angular units.

units

Returns a 2-tuple of the units value and the units name, and will automatically determines whether to return the linear or angular units.

ellisoid

Returns a tuple of the ellipsoid parameters for this spatial reference: (semimajor axis, semiminor axis, and inverse flattening)

semi_major

Returns the semi major axis of the ellipsoid for this spatial reference.

semi_minor

Returns the semi minor axis of the ellipsoid for this spatial reference.

inverse_flattening

Returns the inverse flattening of the ellipsoid for this spatial reference.

geographic

Returns `True` if this spatial reference is geographic (root node is `GEOGCS`).

local

Returns `True` if this spatial reference is local (root node is `LOCAL_CS`).

projected

Returns `True` if this spatial reference is a projected coordinate system (root node is `PROJCS`).

wkt

Returns the WKT representation of this spatial reference.

pretty_wkt

Returns the 'pretty' representation of the WKT.

proj

Returns the PROJ.4 representation for this spatial reference.

proj4

Alias for `SpatialReference.proj`.

xml

Returns the XML representation of this spatial reference.

CoordTransform

class CoordTransform(*source*, *target*)

Represents a coordinate system transform. It is initialized with two `SpatialReference`, representing the source and target coordinate systems, respectively. These objects should be used when performing the same coordinate transformation repeatedly on different geometries:

```
>>> ct = CoordTransform(SpatialReference('WGS84'), SpatialReference('NAD83'))
>>> for feat in layer:
...     geom = feat.geom # getting clone of feature geometry
...     geom.transform(ct) # transforming
```

Settings

GDAL_LIBRARY_PATH A string specifying the location of the GDAL library. Typically, this setting is only used if the GDAL library is in a non-standard location (e.g., `/home/john/lib/libgdal.so`).

Geolocation with GeoIP

Changed in version 1.4: *Please see the release notes*

Note: In Django 1.4, the `GeoIP` object was moved out of `django.contrib.gis.utils` and into its own module, `django.contrib.gis.geoip`. A shortcut is still provided in `utils`, but will be removed in Django 1.6.

The `GeoIP` object is a ctypes wrapper for the MaxMind GeoIP C API. [31] This interface is a BSD-licensed alternative to the GPL-licensed Python GeoIP interface provided by MaxMind.

In order to perform IP-based geolocation, the `GeoIP` object requires the GeoIP C libary and either the GeoIP Country or City datasets in binary format (the CSV files will not work!). These datasets may be downloaded from MaxMind. Grab the `GeoIP.dat.gz` and `GeoLiteCity.dat.gz` and unzip them in a directory corresponding to what you set `GEOIP_PATH` with in your settings. See the example and reference below for more details.

Example

Assuming you have the GeoIP C library installed, here is an example of its usage:

```
>>> from django.contrib.gis.geoip import GeoIP
>>> g = GeoIP()
>>> g.country('google.com')
{'country_code': 'US', 'country_name': 'United States'}
>>> g.city('72.14.207.99')
{'area_code': 650,
'city': 'Mountain View',
'country_code': 'US',
'country_code3': 'USA',
'country_name': 'United States',
'dma_code': 807,
'latitude': 37.419200897216797,
'longitude': -122.05740356445312,
'postal_code': '94043',
'region': 'CA'}
>>> g.lat_lon('salon.com')
(37.789798736572266, -122.39420318603516)
>>> g.lon_lat('uh.edu')
(-95.415199279785156, 29.77549934387207)
>>> g.geos('24.124.1.80').wkt
'POINT (-95.2087020874023438 39.0392990112304688)'
```

`GeoIP` Settings

GEOIP_PATH A string specifying the directory where the GeoIP data files are located. This setting is *required* unless manually specified with `path` keyword when initializing the `GeoIP` object.

GEOIP_LIBRARY_PATH A string specifying the location of the GeoIP C library. Typically, this setting is only used if the GeoIP C library is in a non-standard location (e.g., `/home/sue/lib/libGeoIP.so`).

GEOIP_COUNTRY The basename to use for the GeoIP country data file. Defaults to `'GeoIP.dat'`.

[31] GeoIP(R) is a registered trademark of MaxMind, LLC of Boston, Massachusetts.

GEOIP_CITY The basename to use for the GeoIP city data file. Defaults to `'GeoLiteCity.dat'`.

`GeoIP` API

class `GeoIP`(*[path=None, cache=0, country=None, city=None]*)

The `GeoIP` object does not require any parameters to use the default settings. However, at the very least the `GEOIP_PATH` setting should be set with the path of the location of your GeoIP data sets. The following intialization keywords may be used to customize any of the defaults.

Keyword Arguments	Description
`path`	Base directory to where GeoIP data is located or the full path to where the city or country data files (.dat) are located. Assumes that both the city and country data sets are located in this directory; overrides the `GEOIP_PATH` settings attribute.
`cache`	The cache settings when opening up the GeoIP datasets, and may be an integer in (0, 1, 2, 4) corresponding to the `GEOIP_STANDARD`, `GEOIP_MEMORY_CACHE`, `GEOIP_CHECK_CACHE`, and `GEOIP_INDEX_CACHE` `GeoIPOptions` C API settings, respectively. Defaults to 0 (`GEOIP_STANDARD`).
`country`	The name of the GeoIP country data file. Defaults to `GeoIP.dat`. Setting this keyword overrides the `GEOIP_COUNTRY` settings attribute.
`city`	The name of the GeoIP city data file. Defaults to `GeoLiteCity.dat`. Setting this keyword overrides the `GEOIP_CITY` settings attribute.

`GeoIP` Methods

Querying All the following querying routines may take either a string IP address or a fully qualified domain name (FQDN). For example, both `'205.186.163.125'` and `'djangoproject.com'` would be valid query parameters.

`GeoIP.`**`city`**(*query*)

Returns a dictionary of city information for the given query. Some of the values in the dictionary may be undefined (`None`).

`GeoIP.`**`country`**(*query*)

Returns a dictionary with the country code and country for the given query.

`GeoIP.`**`country_code`**(*query*)

Returns only the country code corresponding to the query.

`GeoIP.`**`country_name`**(*query*)

Returns only the country name corresponding to the query.

Coordinate Retrieval

`GeoIP.`**`coords`**(*query*)
Returns a coordinate tuple of (longitude, latitude).

`GeoIP.`**`lon_lat`**(*query*)

Returns a coordinate tuple of (longitude, latitude).

`GeoIP.`**`lat_lon`**(*query*)

Returns a coordinate tuple of (latitude, longitude),

`GeoIP.`**`geos`**(*query*)

Returns a `django.contrib.gis.geos.Point` object corresponding to the query.

Database Information
`GeoIP.`**`country_info`**
This property returns information about the GeoIP country database.

`GeoIP.`**`city_info`**

This property returns information about the GeoIP city database.

`GeoIP.`**`info`**

This property returns information about all GeoIP databases (both city and country), and the version of the GeoIP C library (if supported).

GeoIP-Python API compatibility methods These methods exist to ease compatibility with any code using MaxMind's existing Python API.

classmethod `GeoIP.`**`open`**(*path*, *cache*)

This classmethod instantiates the GeoIP object from the given database path and given cache setting.

`GeoIP.`**`region_by_addr`**(*query*)

`GeoIP.`**`region_by_name`**(*query*)

`GeoIP.`**`record_by_addr`**(*query*)

`GeoIP.`**`record_by_name`**(*query*)

`GeoIP.`**`country_code_by_addr`**(*query*)

`GeoIP.`**`country_code_by_name`**(*query*)

`GeoIP.`**`country_name_by_addr`**(*query*)

`GeoIP.`**`country_name_by_name`**(*query*)

GeoDjango Utilities

The `django.contrib.gis.utils` module contains various utilities that are useful in creating geospatial Web applications.

`LayerMapping` data import utility

The `LayerMapping` class provides a way to map the contents of vector spatial data files (e.g. shapefiles) into GeoDjango models.

This utility grew out of the author's personal needs to eliminate the code repetition that went into pulling geometries and fields out of a vector layer, converting to another coordinate system (e.g. WGS84), and then inserting into a GeoDjango model.

Note: Use of `LayerMapping` requires GDAL.

Warning: GIS data sources, like shapefiles, may be very large. If you find that `LayerMapping` is using too much memory, set `DEBUG` to `False` in your settings. When `DEBUG` is set to `True`, Django *automatically logs every* SQL query – thus, when SQL statements contain geometries, it is easy to consume more memory than is typical.

Example

1. You need a GDAL-supported data source, like a shapefile (here we're using a simple polygon shapefile, `test_poly.shp`, with three features):

```
>>> from django.contrib.gis.gdal import DataSource
>>> ds = DataSource('test_poly.shp')
>>> layer = ds[0]
>>> print(layer.fields) # Exploring the fields in the layer, we only want the 'str' field.
['float', 'int', 'str']
>>> print(len(layer)) # getting the number of features in the layer (should be 3)
3
>>> print(layer.geom_type) # Should be 'Polygon'
Polygon
>>> print(layer.srs) # WGS84 in WKT
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_1984",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.01745329251994295]]
```

2. Now we define our corresponding Django model (make sure to use `syncdb`):

```
from django.contrib.gis.db import models

class TestGeo(models.Model):
    name = models.CharField(max_length=25) # corresponds to the 'str' field
    poly = models.PolygonField(srid=4269) # we want our model in a different SRID
    objects = models.GeoManager()
    def __unicode__(self):
        return 'Name: %s' % self.name
```

3. Use `LayerMapping` to extract all the features and place them in the database:

```
>>> from django.contrib.gis.utils import LayerMapping
>>> from geoapp.models import TestGeo
>>> mapping = {'name' : 'str', # The 'name' model field maps to the 'str' layer field.
               'poly' : 'POLYGON', # For geometry fields use OGC name.
               } # The mapping is a dictionary
>>> lm = LayerMapping(TestGeo, 'test_poly.shp', mapping)
>>> lm.save(verbose=True) # Save the layermap, imports the data.
Saved: Name: 1
Saved: Name: 2
Saved: Name: 3
```

Here, `LayerMapping` just transformed the three geometries from the shapefile in their original spatial reference system (WGS84) to the spatial reference system of the GeoDjango model (NAD83). If no spatial reference system is defined for the layer, use the `source_srs` keyword with a `SpatialReference` object to specify one.

`LayerMapping` API

class `LayerMapping`(*model*, *data_source*, *mapping*[, *layer=0*, *source_srs=None*, *encoding=None*, *transaction_mode='commit_on_success'*, *transform=True*, *unique=True*, *using='default'*])

The following are the arguments and keywords that may be used during instantiation of `LayerMapping` objects.

Argument	Description
`model`	The geographic model, *not* an instance.
`data_source`	The path to the OGR-supported data source file (e.g., a shapefile). Also accepts `django.contrib.gis.gdal.DataSource` instances.
`mapping`	A dictionary: keys are strings corresponding to the model field, and values correspond to string field names for the OGR feature, or if the model field is a geographic then it should correspond to the OGR geometry type, e.g., `'POINT'`, `'LINESTRING'`, `'POLYGON'`.

Keyword Arguments	
`layer`	The index of the layer to use from the Data Source (defaults to 0)
`source_srs`	Use this to specify the source SRS manually (for example, some shapefiles don't come with a '.prj' file). An integer SRID, WKT or PROJ.4 strings, and `django.contrib.gis.gdal.SpatialReference` objects are accepted.
`encoding`	Specifies the character set encoding of the strings in the OGR data source. For example, `'latin-1'`, `'utf-8'`, and `'cp437'` are all valid encoding parameters.
`transaction_mode`	May be `'commit_on_success'` (default) or `'autocommit'`.
`transform`	Setting this to False will disable coordinate transformations. In other words, geometries will be inserted into the database unmodified from their original state in the data source.
`unique`	Setting this to the name, or a tuple of names, from the given model will create models unique only to the given name(s). Geometries will from each feature will be added into the collection associated with the unique model. Forces the transaction mode to be `'autocommit'`.
`using`	New in version 1.2. Sets the database to use when importing spatial data. Default is `'default'`

`save()` Keyword Arguments

`LayerMapping.`**`save`**`(`[*verbose=False*, *fid_range=False*, *step=False*, *progress=False*, *silent=False*, *stream=sys.stdout*, *strict=False*]`)`

The `save()` method also accepts keywords. These keywords are used for controlling output logging, error handling, and for importing specific feature ranges.

Save Keyword Arguments	Description
`fid_range`	May be set with a slice or tuple of (begin, end) feature ID's to map from the data source. In other words, this keyword enables the user to selectively import a subset range of features in the geographic data source.
`progress`	When this keyword is set, status information will be printed giving the number of features processed and successfully saved. By default, progress information will be printed every 1000 features processed, however, this default may be overridden by setting this keyword with an integer for the desired interval.
`silent`	By default, non-fatal error notifications are printed to `sys.stdout`, but this keyword may be set to disable these notifications.
`step`	If set with an integer, transactions will occur at every step interval. For example, if `step=1000`, a commit would occur after the 1,000th feature, the 2,000th feature etc.
`stream`	Status information will be written to this file handle. Defaults to using `sys.stdout`, but any object with a `write` method is supported.
`strict`	Execution of the model mapping will cease upon the first error encountered. The default value (`False`) behavior is to attempt to continue.
`verbose`	If set, information will be printed subsequent to each model save executed on the database.

Troubleshooting

Running out of memory As noted in the warning at the top of this section, Django stores all SQL queries when `DEBUG=True`. Set `DEBUG=False` in your settings, and this should stop excessive memory use when running `LayerMapping` scripts.

MySQL: `max_allowed_packet` error If you encounter the following error when using `LayerMapping` and MySQL:

```
OperationalError: (1153, "Got a packet bigger than 'max_allowed_packet' bytes")
```

Then the solution is to increase the value of the `max_allowed_packet` setting in your MySQL configuration. For example, the default value may be something low like one megabyte – the setting may be modified in MySQL's configuration file (`my.cnf`) in the `[mysqld]` section:

```
max_allowed_packet = 10M
```

OGR Inspection

ogrinspect
ogrinspect(*data_source*, *model_name*[, ***kwargs*])

mapping
mapping(*data_source*[, *geom_name='geom'*, *layer_key=0*, *multi_geom=False*])

GeoDjango Management Commands

inspectdb

django-admin.py inspectdb

When `django.contrib.gis` is in your `INSTALLED_APPS`, the `inspectdb` management command is overridden with one from GeoDjango. The overridden command is spatially-aware, and places geometry fields in the auto-generated model definition, where appropriate.

ogrinspect <data_source> <model_name>

django-admin.py ogrinspect

The `ogrinpsect` management command will inspect the given OGR-compatible `DataSource` (e.g., a shapefile) and will output a GeoDjango model with the given model name. There's a detailed example of using `ogrinspect` *in the tutorial*.

-blank <blank_field(s)>
: Use a comma separated list of OGR field names to add the `blank=True` keyword option to the field definition. Set with `true` to apply to all applicable fields.

-decimal <decimal_field(s)>
: Use a comma separated list of OGR float fields to generate `DecimalField` instead of the default `FloatField`. Set to `true` to apply to all OGR float fields.

-geom-name <name>
: Specifies the model attribute name to use for the geometry field. Defaults to `'geom'`.

-layer <layer>
The key for specifying which layer in the OGR `DataSource` source to use. Defaults to 0 (the first layer). May be an integer or a string identifier for the `Layer`.

-mapping
Automatically generate a mapping dictionary for use with `LayerMapping`.

-multi-geom
When generating the geometry field, treat it as a geometry collection. For example, if this setting is enabled then a `MultiPolygonField` will be placed in the generated model rather than `PolygonField`.

-name-field <name_field>
Generates a `__unicode__` routine on the model that will return the the given field name.

-no-imports
Suppresses the `from django.contrib.gis.db import models` import statement.

-null <null_field(s)>
Use a comma separated list of OGR field names to add the `null=True` keyword option to the field definition. Set with `true` to apply to all applicable fields.

-srid
The SRID to use for the geometry field. If not set, `ogrinspect` attempts to automatically determine of the SRID of the data source.

GeoDjango's admin site

GeoModelAdmin

class GeoModelAdmin

default_lon

The default center longitude.

default_lat

The default center latitude.

default_zoom

The default zoom level to use. Defaults to 18.

extra_js

Sequence of URLs to any extra JavaScript to include.

map_template

Override the template used to generate the JavaScript slippy map. Default is `'gis/admin/openlayers.html'`.

map_width

Width of the map, in pixels. Defaults to 600.

map_height

Height of the map, in pixels. Defaults to 400.

openlayers_url

Link to the URL of the OpenLayers JavaScript. Defaults to `'http://openlayers.org/api/2.8/OpenLayers.js'`.

modifiable

Defaults to `True`. When set to `False`, disables editing of existing geometry fields in the admin.

Note: This is different from adding the geometry field to `readonly_fields`, which will only display the WKT of the geometry. Setting `modifiable=False`, actually displays the geometry in a map, but disables the ability to edit its vertices.

OSMGeoAdmin

class OSMGeoAdmin

A subclass of `GeoModelAdmin` that uses a spherical mercator projection with OpenStreetMap street data tiles. See the *OSMGeoAdmin introduction* in the tutorial for a usage example.

Geographic Feeds

GeoDjango has its own `Feed` subclass that may embed location information in RSS/Atom feeds formatted according to either the Simple GeoRSS or W3C Geo standards. Because GeoDjango's syndication API is a superset of Django's, please consult *Django's syndication documentation* for details on general usage.

Example

API Reference

Feed Subclass

class Feed

In addition to methods provided by the `django.contrib.syndication.feeds.Feed` base class, GeoDjango's `Feed` class provides the following overrides. Note that these overrides may be done in multiple ways:

```
from django.contrib.gis.feeds import Feed

class MyFeed(Feed):

    # First, as a class attribute.
    geometry = ...
    item_geometry = ...

    # Also a function with no arguments
    def geometry(self):
        ...

    def item_geometry(self):
        ...

    # And as a function with a single argument
    def geometry(self, obj):
        ...
```

```
def item_geometry(self, item):
    ...
```

geometry(*obj*)

Takes the object returned by `get_object()` and returns the *feed's* geometry. Typically this is a `GEOSGeometry` instance, or can be a tuple to represent a point or a box. For example:

```
class ZipcodeFeed(Feed):

    def geometry(self, obj):
        # Can also return: 'obj.poly', and 'obj.poly.centroid'.
        return obj.poly.extent # tuple like: (X0, Y0, X1, Y1).
```

item_geometry(*item*)

Set this to return the geometry for each *item* in the feed. This can be a `GEOSGeometry` instance, or a tuple that represents a point coordinate or bounding box. For example:

```
class ZipcodeFeed(Feed):

    def item_geometry(self, obj):
        # Returns the polygon.
        return obj.poly
```

SyndicationFeed Subclasses The following `django.utils.feedgenerator.SyndicationFeed` subclasses are available:

class GeoRSSFeed

class GeoAtom1Feed

class W3CGeoFeed

Note: W3C Geo formatted feeds only support `PointField` geometries.

Geographic Sitemaps

Google's sitemap protocol used to include geospatial content support. [32] This included the addition of the `<url>` child element `<geo:geo>`, which tells Google that the content located at the URL is geographic in nature. This is now obsolete.

Example

Reference

KMLSitemap

KMZSitemap

GeoRSSSitemap

[32] Google, Inc., What is a Geo Sitemap?.

Testing GeoDjango apps

Included in this documentation are some additional notes and settings for *PostGIS* and *SpatiaLite* users.

PostGIS

Settings

Note: The settings below have sensible defaults, and shouldn't require manual setting.

POSTGIS_TEMPLATE This setting may be used to customize the name of the PostGIS template database to use. It automatically defaults to `'template_postgis'` (the same name used in the *installation documentation*).

POSTGIS_VERSION When GeoDjango's spatial backend initializes on PostGIS, it has to perform a SQL query to determine the version in order to figure out what features are available. Advanced users wishing to prevent this additional query may set the version manually using a 3-tuple of integers specifying the major, minor, and subminor version numbers for PostGIS. For example, to configure for PostGIS 1.5.2 you would use:

```
POSTGIS_VERSION = (1, 5, 2)
```

Obtaining sufficient privileges Depending on your configuration, this section describes several methods to configure a database user with sufficient privileges to run tests for GeoDjango applications on PostgreSQL. If your *spatial database template* was created like in the instructions, then your testing database user only needs to have the ability to create databases. In other configurations, you may be required to use a database superuser.

Create database user To make a database user with the ability to create databases, use the following command:

```
$ createuser --createdb -R -S <user_name>
```

The `-R -S` flags indicate that we do not want the user to have the ability to create additional users (roles) or to be a superuser, respectively.

Alternatively, you may alter an existing user's role from the SQL shell (assuming this is done from an existing superuser account):

```
postgres# ALTER ROLE <user_name> CREATEDB NOSUPERUSER NOCREATEROLE;
```

Create database superuser This may be done at the time the user is created, for example:

```
$ createuser --superuser <user_name>
```

Or you may alter the user's role from the SQL shell (assuming this is done from an existing superuser account):

```
postgres# ALTER ROLE <user_name> SUPERUSER;
```

Create local PostgreSQL database

1. Initialize database: `initdb -D /path/to/user/db`
2. If there's already a Postgres instance on the machine, it will need to use a different TCP port than 5432. Edit `postgresql.conf` (in `/path/to/user/db`) to change the database port (e.g. `port = 5433`).
3. Start this database `pg_ctl -D /path/to/user/db start`

Windows On Windows platforms the pgAdmin III utility may also be used as a simple way to add superuser privileges to your database user.

By default, the PostGIS installer on Windows includes a template spatial database entitled `template_postgis`.

SpatiaLite

Make sure the necessary spatial tables are created in your test spatial database, as described in *Creating a spatial database for SpatiaLite*. Then just do this:

```
$ python manage.py test
```

Settings

`SPATIALITE_SQL` Only relevant when using a SpatiaLite version older than 3.0.

By default, the GeoDjango test runner looks for the SpatiaLite SQL in the same directory where it was invoked (by default the same directory where `manage.py` is located). To use a different location, add the following to your settings:

```
SPATIALITE_SQL='/path/to/init_spatialite-2.3.sql'
```

GeoDjango tests

Changed in version 1.3: *Please see the release notes* GeoDjango's test suite may be run in one of two ways, either by itself or with the rest of *Django's unit tests*.

Run only GeoDjango tests To run *only* the tests for GeoDjango, the `TEST_RUNNER` setting must be changed to use the `GeoDjangoTestSuiteRunner`:

```
TEST_RUNNER = 'django.contrib.gis.tests.GeoDjangoTestSuiteRunner'
```

Example First, you'll need a bare-bones settings file, like below, that is customized with your spatial database name and user:

```
TEST_RUNNER = 'django.contrib.gis.tests.GeoDjangoTestSuiteRunner'

DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'a_spatial_database',
        'USER': 'db_user'
    }
}
```

Assuming the above is in a file called `postgis.py` that is in the the same directory as `manage.py` of your Django project, then you may run the tests with the following command:

```
$ python manage.py test --settings=postgis
```

Run with `runtests.py` To have the GeoDjango tests executed when *running the Django test suite* with `runtests.py` all of the databases in the settings file must be using one of the *spatial database backends*.

Warning: Do not change the `TEST_RUNNER` setting when running the GeoDjango tests with `runtests.py`.

Example The following is an example bare-bones settings file with spatial backends that can be used to run the entire Django test suite, including those in `django.contrib.gis`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'geodjango',
        'USER': 'geodjango',
    },
   'other': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'other',
        'USER': 'geodjango',
   }
}
```

Assuming the settings above were in a `postgis.py` file in the same directory as `runtests.py`, then all Django and GeoDjango tests would be performed when executing the command:

```
$ ./runtests.py --settings=postgis
```

Deploying GeoDjango

Basically, the deployment of a GeoDjango application is not different from the deployment of a normal Django application. Please consult Django's *deployment documentation*.

Warning: GeoDjango uses the GDAL geospatial library which is not thread safe at this time. Thus, it is *highly* recommended to not use threading when deploying – in other words, use an appropriate configuration of Apache or the prefork method when using FastCGI through another Web server.
For example, when configuring your application with `mod_wsgi`, set the `WSGIDaemonProcess` attribute `threads` to `1`, unless Apache may crash when running your GeoDjango application. Increase the number of `processes` instead.

6.4.10 django.contrib.humanize

A set of Django template filters useful for adding a "human touch" to data.

To activate these filters, add `'django.contrib.humanize'` to your `INSTALLED_APPS` setting. Once you've done that, use `{% load humanize %}` in a template, and you'll have access to the following filters.

apnumber

For numbers 1-9, returns the number spelled out. Otherwise, returns the number. This follows Associated Press style.

Examples:

- `1` becomes `one`.

- `2` becomes `two`.
- `10` becomes `10`.

You can pass in either an integer or a string representation of an integer.

intcomma

Converts an integer to a string containing commas every three digits.

Examples:

- `4500` becomes `4,500`.
- `45000` becomes `45,000`.
- `450000` becomes `450,000`.
- `4500000` becomes `4,500,000`.

Format localization will be respected if enabled, e.g. with the `'de'` language:

- `45000` becomes `'45.000'`.
- `450000` becomes `'450.000'`.

You can pass in either an integer or a string representation of an integer.

intword

Converts a large integer to a friendly text representation. Works best for numbers over 1 million.

Examples:

- `1000000` becomes `1.0 million`.
- `1200000` becomes `1.2 million`.
- `1200000000` becomes `1.2 billion`.

Values up to 10^100 (Googol) are supported.

Format localization will be respected if enabled, e.g. with the `'de'` language:

- `1000000` becomes `'1,0 Million'`.
- `1200000` becomes `'1,2 Million'`.
- `1200000000` becomes `'1,2 Milliarden'`.

You can pass in either an integer or a string representation of an integer.

naturalday

For dates that are the current day or within one day, return “today”, “tomorrow” or “yesterday”, as appropriate. Otherwise, format the date using the passed in format string.

Argument: Date formatting string as described in the `date` tag.

Examples (when ‘today’ is 17 Feb 2007):

- `16 Feb 2007` becomes `yesterday`.
- `17 Feb 2007` becomes `today`.

- `18 Feb 2007` becomes `tomorrow`.
- Any other day is formatted according to given argument or the `DATE_FORMAT` setting if no argument is given.

naturaltime

New in version 1.4: *Please see the release notes* For datetime values, returns a string representing how many seconds, minutes or hours ago it was – falling back to a longer date format if the value is more than a day old. In case the datetime value is in the future the return value will automatically use an appropriate phrase.

Examples (when 'now' is 17 Feb 2007 16:30:00):

- `17 Feb 2007 16:30:00` becomes `now`.
- `17 Feb 2007 16:29:31` becomes `29 seconds ago`.
- `17 Feb 2007 16:29:00` becomes `a minute ago`.
- `17 Feb 2007 16:25:35` becomes `4 minutes ago`.
- `17 Feb 2007 15:30:29` becomes `an hour ago`.
- `17 Feb 2007 13:31:29` becomes `2 hours ago`.
- `16 Feb 2007 13:31:29` becomes `1 day ago`.
- `17 Feb 2007 16:30:30` becomes `29 seconds from now`.
- `17 Feb 2007 16:31:00` becomes `a minute from now`.
- `17 Feb 2007 16:34:35` becomes `4 minutes from now`.
- `17 Feb 2007 16:30:29` becomes `an hour from now`.
- `17 Feb 2007 18:31:29` becomes `2 hours from now`.
- `18 Feb 2007 16:31:29` becomes `1 day from now`.

ordinal

Converts an integer to its ordinal as a string.

Examples:

- `1` becomes `1st`.
- `2` becomes `2nd`.
- `3` becomes `3rd`.

You can pass in either an integer or a string representation of an integer.

6.4.11 The "local flavor" add-ons

Following its "batteries included" philosophy, Django comes with assorted pieces of code that are useful for particular countries or cultures. These are called the "local flavor" add-ons and live in the `django.contrib.localflavor` package.

Inside that package, country- or culture-specific code is organized into subpackages, named using ISO 3166 country codes.

Most of the `localflavor` add-ons are localized form components deriving from the *forms* framework – for example, a `USStateField` that knows how to validate U.S. state abbreviations, and a `FISocialSecurityNumber` that knows how to validate Finnish social security numbers.

To use one of these localized components, just import the relevant subpackage. For example, here's how you can create a form with a field representing a French telephone number:

```
from django import forms
from django.contrib.localflavor.fr.forms import FRPhoneNumberField

class MyForm(forms.Form):
    my_french_phone_no = FRPhoneNumberField()
```

Supported countries

Countries currently supported by `localflavor` are:

- Argentina
- Australia
- Austria
- Belgium
- Brazil
- Canada
- Chile
- China
- Colombia
- Croatia
- Czech
- Ecuador
- Finland
- France
- Germany
- Hong Kong
- Iceland
- India
- Indonesia
- Ireland
- Israel
- Italy
- Japan
- Kuwait
- Macedonia

- Mexico
- The Netherlands
- Norway
- Peru
- Poland
- Portugal
- Paraguay
- Romania
- Russia
- Slovakia
- Slovenia
- South Africa
- Spain
- Sweden
- Switzerland
- Turkey
- United Kingdom
- United States of America
- Uruguay

The `django.contrib.localflavor` package also includes a `generic` subpackage, containing useful code that is not specific to one particular country or culture. Currently, it defines date, datetime and split datetime input fields based on those from *forms*, but with non-US default formats. Here's an example of how to use them:

```
from django import forms
from django.contrib.localflavor import generic

class MyForm(forms.Form):
    my_date_field = generic.forms.DateField()
```

Internationalization of localflavor

Localflavor has its own catalog of translations, in the directory `django/contrib/localflavor/locale`, and it's not loaded automatically like Django's general catalog in `django/conf/locale`. If you want localflavor's texts to be translated, like form fields error messages, you must include `django.contrib.localflavor` in the `INSTALLED_APPS` setting, so the internationalization system can find the catalog, as explained in *How Django discovers translations*.

Adding flavors

We'd love to add more of these to Django, so please create a ticket with any code you'd like to contribute. One thing we ask is that you please use Unicode objects (`u'mystring'`) for strings, rather than setting the encoding in the file. See any of the existing flavors for examples.

Localflavor and backwards compatibility

As documented in our *API stability* policy, Django will always attempt to make `django.contrib.localflavor` reflect the officially gazetted policies of the appropriate local government authority. For example, if a government body makes a change to add, alter, or remove a province (or state, or county), that change will be reflected in Django's localflavor in the next stable Django release.

When a backwards-incompatible change is made (for example, the removal or renaming of a province) the localflavor in question will raise a warning when that localflavor is imported. This provides a runtime indication that something may require attention.

However, once you have addressed the backwards compatibility (for example, auditing your code to see if any data migration is required), the warning serves no purpose. The warning can then be supressed. For example, to suppress the warnings raised by the Indonesian localflavor you would use the following code:

```
import warnings
warnings.filterwarnings('ignore',
                        category=RuntimeWarning,
                        module='django.contrib.localflavor.id')
from django.contrib.localflavor.id import forms as id_forms
```

Argentina (`ar`)

class `ar.forms.`**`ARPostalCodeField`**
A form field that validates input as either a classic four-digit Argentinian postal code or a CPA.

class `ar.forms.`**`ARDNIField`**
A form field that validates input as a Documento Nacional de Identidad (DNI) number.

class `ar.forms.`**`ARCUITField`**
A form field that validates input as a Codigo Unico de Identificacion Tributaria (CUIT) number.

class `ar.forms.`**`ARProvinceSelect`**
A `Select` widget that uses a list of Argentina's provinces and autonomous cities as its choices.

Australia (`au`)

New in version 1.4: *Please see the release notes*

class `au.forms.`**`AUPostCodeField`**
A form field that validates input as an Australian postcode.

class `au.forms.`**`AUPhoneNumberField`**
A form field that validates input as an Australian phone number. Valid numbers have ten digits.

class `au.forms.`**`AUStateSelect`**
A `Select` widget that uses a list of Australian states/territories as its choices.

class `au.models.`**`AUPhoneNumberField`**
A model field that checks that the value is a valid Australian phone number (ten digits).

class `au.models.`**`AUStateField`**
A model field that forms represent as a `forms.AUStateField` field and stores the three-letter Australian state abbreviation in the database.

class `au.models.`**`AUPostCodeField`**
A model field that forms represent as a `forms.AUPostCodeField` field and stores the four-digit Australian postcode in the database.

Austria (at)

class at.forms.**ATZipCodeField**
A form field that validates its input as an Austrian zip code, with the format XXXX (first digit must be greater than 0).

class at.forms.**ATStateSelect**
A Select widget that uses a list of Austrian states as its choices.

class at.forms.**ATSocialSecurityNumberField**
A form field that validates its input as an Austrian social security number.

Belgium (be)

New in version 1.3: *Please see the release notes*

class be.forms.**BEPhoneNumberField**
A form field that validates input as a Belgium phone number, with one of the formats 0x xxx xx xx, 0xx xx xx xx, 04xx xx xx xx, 0x/xxx.xx.xx, 0xx/xx.xx.xx, 04xx/xx.xx.xx, 0x.xxx.xx.xx, 0xx.xx.xx.xx, 04xx.xx.xx.xx, 0xxxxxxxx or 04xxxxxxxx.

class be.forms.**BEPostalCodeField**
A form field that validates input as a Belgium postal code, in the range and format 1XXX-9XXX.

class be.forms.**BEProvinceSelect**
A Select widget that uses a list of Belgium provinces as its choices.

class be.forms.**BERegionSelect**
A Select widget that uses a list of Belgium regions as its choices.

Brazil (br)

class br.forms.**BRPhoneNumberField**
A form field that validates input as a Brazilian phone number, with the format XX-XXXX-XXXX.

class br.forms.**BRZipCodeField**
A form field that validates input as a Brazilian zip code, with the format XXXXX-XXX.

class br.forms.**BRStateSelect**
A Select widget that uses a list of Brazilian states/territories as its choices.

class br.forms.**BRCPFField**
A form field that validates input as Brazilian CPF.

Input can either be of the format XXX.XXX.XXX-VD or be a group of 11 digits.

class br.forms.**BRCNPJField**
A form field that validates input as Brazilian CNPJ.

Input can either be of the format XX.XXX.XXX/XXXX-XX or be a group of 14 digits.

Canada (ca)

class ca.forms.**CAPhoneNumberField**
A form field that validates input as a Canadian phone number, with the format XXX-XXX-XXXX.

class ca.forms.**CAPostalCodeField**
A form field that validates input as a Canadian postal code, with the format XXX XXX.

class ca.forms.**CAProvinceField**
A form field that validates input as a Canadian province name or abbreviation.

class ca.forms.**CASocialInsuranceNumberField**
A form field that validates input as a Canadian Social Insurance Number (SIN). A valid number must have the format XXX-XXX-XXX and pass a Luhn mod-10 checksum.

class ca.forms.**CAProvinceSelect**
A Select widget that uses a list of Canadian provinces and territories as its choices.

Chile (cl)

class cl.forms.**CLRutField**
A form field that validates input as a Chilean national identification number ('Rol Unico Tributario' or RUT). The valid format is XX.XXX.XXX-X.

class cl.forms.**CLRegionSelect**
A Select widget that uses a list of Chilean regions (Regiones) as its choices.

China (cn)

New in version 1.4: *Please see the release notes*

class cn.forms.**CNProvinceSelect**
A Select widget that uses a list of Chinese regions as its choices.

class cn.forms.**CNPostCodeField**
A form field that validates input as a Chinese post code. Valid formats are XXXXXX where X is digit.

class cn.forms.**CNIDCardField**
A form field that validates input as a Chinese Identification Card Number. Both 1st and 2nd generation ID Card Number are validated.

class cn.forms.**CNPhoneNumberField**
A form field that validates input as a Chinese phone number. Valid formats are 0XX-XXXXXXXX, composed of 3 or 4 digits of region code and 7 or 8 digits of phone number.

class cn.forms.**CNCellNumberField**
A form field that validates input as a Chinese mobile phone number. Valid formats are like 1XXXXXXXXXX, where X is digit. The second digit could only be 3, 5 and 8.

Colombia (co)

New in version 1.4: *Please see the release notes*

class co.forms.**CoDepartmentSelect**
A Select widget that uses a list of Colombian departments as its choices.

Croatia (hr)

New in version 1.4: *Please see the release notes*

class hr.forms.**HRCountySelect**
A Select widget that uses a list of counties of Croatia as its choices.

class hr.forms.**HRPhoneNumberPrefixSelect**
A Select widget that uses a list of phone number prefixes of Croatia as its choices.

class hr.forms.**HRLicensePlatePrefixSelect**
A Select widget that uses a list of vehicle license plate prefixes of Croatia as its choices.

class hr.forms.**HRPhoneNumberField**
A form field that validates input as a phone number of Croatia. A valid format is a country code or a leading zero, area code prefix, 6 or 7 digit number; e.g. +385XXXXXXXX or 0XXXXXXXX Validates fixed, mobile and FGSM numbers. Normalizes to a full number with country code (+385 prefix).

class hr.forms.**HRLicensePlateField**
A form field that validates input as a vehicle license plate of Croatia. Normalizes to the specific format XX YYYY-XX where X is a letter and Y a digit. There can be three or four digits. Suffix is constructed from the shared letters of the Croatian and English alphabets. It is used for standardized license plates only. Special cases like license plates for oldtimers, temporary license plates, government institution license plates and customized license plates are not covered by this field.

class hr.forms.**HRPostalCodeField**
A form field that validates input as a postal code of Croatia. It consists of exactly five digits ranging from 10000 to 59999 inclusive.

class hr.forms.**HROIBField**
A form field that validates input as a Personal Identification Number (OIB) of Croatia. It consists of exactly eleven digits.

class hr.forms.**HRJMBGField**
A form field that validates input as a Unique Master Citizen Number (JMBG). The number is still in use in Croatia, but it is being replaced by OIB. This field works for other ex-Yugoslavia countries as well where the JMBG is still in use. The area segment of the JMBG is not validated because the citizens might have emigrated to another ex-Yugoslavia country. The number consists of exactly thirteen digits.

class hr.forms.**HRJMBAGField**
A form field that validates input as a Unique Master Academic Citizen Number (JMBAG) of Croatia. This number is used by college students and professors in Croatia. The number consists of exactly nineteen digits.

Czech (cz)

class cz.forms.**CZPostalCodeField**
A form field that validates input as a Czech postal code. Valid formats are XXXXX or XXX XX, where X is a digit.

class cz.forms.**CZBirthNumberField**
A form field that validates input as a Czech Birth Number. A valid number must be in format XXXXXX/XXXX (slash is optional).

class cz.forms.**CZICNumberField**
A form field that validates input as a Czech IC number field.

class cz.forms.**CZRegionSelect**
A Select widget that uses a list of Czech regions as its choices.

Ecuador (ec)

New in version 1.4: *Please see the release notes*

class ec.forms.**EcProvinceSelect**
A Select widget that uses a list of Ecuatorian provinces as its choices.

Finland (fi)

class fi.forms.**FISocialSecurityNumber**
A form field that validates input as a Finnish social security number.

class fi.forms.**FIZipCodeField**
A form field that validates input as a Finnish zip code. Valid codes consist of five digits.

class fi.forms.**FIMunicipalitySelect**
A Select widget that uses a list of Finnish municipalities as its choices.

France (fr)

class fr.forms.**FRPhoneNumberField**
A form field that validates input as a French local phone number. The correct format is 0X XX XX XX XX. 0X.XX.XX.XX.XX and 0XXXXXXXXX validate but are corrected to 0X XX XX XX XX.

class fr.forms.**FRZipCodeField**
A form field that validates input as a French zip code. Valid codes consist of five digits.

class fr.forms.**FRDepartmentSelect**
A Select widget that uses a list of French departments as its choices.

Germany (de)

class de.forms.**DEIdentityCardNumberField**
A form field that validates input as a German identity card number (Personalausweis). Valid numbers have the format XXXXXXXXXXX-XXXXXXX-XXXXXXX-X, with no group consisting entirely of zeroes.

class de.forms.**DEZipCodeField**
A form field that validates input as a German zip code. Valid codes consist of five digits.

class de.forms.**DEStateSelect**
A Select widget that uses a list of German states as its choices.

Hong Kong (hk)

class hk.forms.**HKPhoneNumberField**
A form field that validates input as a Hong Kong phone number.

The Netherlands (nl)

class nl.forms.**NLPhoneNumberField**
A form field that validates input as a Dutch telephone number.

class nl.forms.**NLSofiNumberField**
A form field that validates input as a Dutch social security number (SoFI/BSN).

class nl.forms.**NLZipCodeField**
A form field that validates input as a Dutch zip code.

class nl.forms.**NLProvinceSelect**
A Select widget that uses a list of Dutch provinces as its list of choices.

Iceland (is_)

class is_.forms.**ISIdNumberField**
A form field that validates input as an Icelandic identification number (kennitala). The format is XXXXXX-XXXX.

class is_.forms.**ISPhoneNumberField**
A form field that validates input as an Icelandtic phone number (seven digits with an optional hyphen or space after the first three digits).

class is_.forms.**ISPostalCodeSelect**
A Select widget that uses a list of Icelandic postal codes as its choices.

India (in_)

class in_.forms.**INStateField**
A form field that validates input as an Indian state/territory name or abbreviation. Input is normalized to the standard two-letter vehicle registration abbreviation for the given state or territory.

class in_.forms.**INZipCodeField**
A form field that validates input as an Indian zip code, with the format XXXXXXX.

class in_.forms.**INStateSelect**
A Select widget that uses a list of Indian states/territories as its choices.

New in version 1.4: *Please see the release notes*

class in_.forms.**INPhoneNumberField**
A form field that validates that the data is a valid Indian phone number, including the STD code. It's normalised to 0XXX-XXXXXXX or 0XXX XXXXXXX format. The first string is the STD code which is a '0' followed by 2-4 digits. The second string is 8 digits if the STD code is 3 digits, 7 digits if the STD code is 4 digits and 6 digits if the STD code is 5 digits. The second string will start with numbers between 1 and 6. The separator is either a space or a hyphen.

Ireland (ie)

class ie.forms.**IECountySelect**
A Select widget that uses a list of Irish Counties as its choices.

Indonesia (id)

class id.forms.**IDPostCodeField**
A form field that validates input as an Indonesian post code field.

class id.forms.**IDProvinceSelect**
A Select widget that uses a list of Indonesian provinces as its choices.

Changed in version 1.3: The province "Nanggroe Aceh Darussalam (NAD)" has been removed from the province list in favor of the new official designation "Aceh (ACE)".

class id.forms.**IDPhoneNumberField**
A form field that validates input as an Indonesian telephone number.

class id.forms.**IDLicensePlatePrefixSelect**
A Select widget that uses a list of Indonesian license plate prefix code as its choices.

class id.forms.**IDLicensePlateField**
A form field that validates input as an Indonesian vehicle license plate.

class id.forms.**IDNationalIdentityNumberField**
A form field that validates input as an Indonesian national identity number (NIK/KTP). The output will be in the format of 'XX.XXXX.DDMMYY.XXXX'. Dots or spaces can be used in the input to break down the numbers.

Israel (il)

class il.forms.**ILPostalCodeField**
A form field that validates its input as an Israeli five-digit postal code.

class il.forms.**ILIDNumberField**
A form field that validates its input as an Israeli identification number. The output will be in the format of a 2-9 digit number, consisting of a 1-8 digit ID number followed by a single checksum digit, calculated using the Luhn algorithm.

Input may contain an optional hyphen separating the ID number from the checksum digit.

Italy (it)

class it.forms.**ITSocialSecurityNumberField**
A form field that validates input as an Italian social security number (codice fiscale).

class it.forms.**ITVatNumberField**
A form field that validates Italian VAT numbers (partita IVA).

class it.forms.**ITZipCodeField**
A form field that validates input as an Italian zip code. Valid codes must have five digits.

class it.forms.**ITProvinceSelect**
A Select widget that uses a list of Italian provinces as its choices.

class it.forms.**ITRegionSelect**
A Select widget that uses a list of Italian regions as its choices.

Japan (jp)

class jp.forms.**JPPostalCodeField**
A form field that validates input as a Japanese postcode. It accepts seven digits, with or without a hyphen.

class jp.forms.**JPPrefectureSelect**
A Select widget that uses a list of Japanese prefectures as its choices.

Kuwait (kw)

class kw.forms.**KWCivilIDNumberField**
A form field that validates input as a Kuwaiti Civil ID number. A valid Civil ID number must obey the following rules:

- The number consist of 12 digits.
- The birthdate of the person is a valid date.
- The calculated checksum equals to the last digit of the Civil ID.

Macedonia (mk)

New in version 1.4: *Please see the release notes*

class mk.forms.**MKIdentityCardNumberField**
A form field that validates input as a Macedonian identity card number. Both old and new identity card numbers are supported.

class mk.forms.**MKMunicipalitySelect**
A form Select widget that uses a list of Macedonian municipalities as choices.

class mk.forms.**UMCNField**
A form field that validates input as a unique master citizen number.

The format of the unique master citizen number is not unique to Macedonia. For more information see: https://secure.wikimedia.org/wikipedia/en/wiki/Unique_Master_Citizen_Number

A value will pass validation if it complies to the following rules:

•Consists of exactly 13 digits

•The first 7 digits represent a valid past date in the format DDMMYYY

•The last digit of the UMCN passes a checksum test

class mk.models.**MKIdentityCardNumberField**
A model field that forms represent as a forms.MKIdentityCardNumberField field.

class mk.models.**MKMunicipalityField**
A model field that forms represent as a forms.MKMunicipalitySelect and stores the 2 character code of the municipality in the database.

class mk.models.**UMCNField**
A model field that forms represent as a forms.UMCNField field.

Mexico (mx)

class mx.forms.**MXZipCodeField**
New in version 1.4: *Please see the release notes* A form field that accepts a Mexican Zip Code.

More info about this: List of postal codes in Mexico (zipcodes)

class mx.forms.**MXRFCField**
New in version 1.4: *Please see the release notes* A form field that validates a Mexican *Registro Federal de Contribuyentes* for either **Persona física** or **Persona moral**. This field accepts RFC strings whether or not it contains a *homoclave*.

More info about this: Registro Federal de Contribuyentes (rfc)

class mx.forms.**MXCURPField**
New in version 1.4: *Please see the release notes* A field that validates a Mexican *Clave Única de Registro de Población*.

More info about this: Clave Unica de Registro de Poblacion (curp)

class mx.forms.**MXStateSelect**
A Select widget that uses a list of Mexican states as its choices.

class mx.models.**MXStateField**
New in version 1.4: *Please see the release notes* A model field that stores the three-letter Mexican state abbreviation in the database.

class mx.models.**MXZipCodeField**
New in version 1.4: *Please see the release notes* A model field that forms represent as a forms.MXZipCodeField field and stores the five-digit Mexican zip code.

class mx.models.**MXRFCField**
New in version 1.4: *Please see the release notes* A model field that forms represent as a forms.MXRFCField field and stores the value of a valid Mexican RFC.

class mx.models.**MXCURPField**
New in version 1.4: *Please see the release notes* A model field that forms represent as a forms.MXCURPField field and stores the value of a valid Mexican CURP.

Additionally, a choice tuple is provided in django.contrib.localflavor.mx.mx_states, allowing customized model and form fields, and form presentations, for subsets of Mexican states abbreviations:

mx.mx_states.**STATE_CHOICES**
A tuple of choices of the states abbreviations for all 31 Mexican states, plus the *Distrito Federal*.

Norway (no)

class no.forms.**NOSocialSecurityNumber**
A form field that validates input as a Norwegian social security number (personnummer).

class no.forms.**NOZipCodeField**
A form field that validates input as a Norwegian zip code. Valid codes have four digits.

class no.forms.**NOMunicipalitySelect**
A Select widget that uses a list of Norwegian municipalities (fylker) as its choices.

Paraguay (py)

New in version 1.4: *Please see the release notes*

class py.forms.**PyDepartmentSelect**
A Select widget with a list of Paraguayan departments as choices.

class py.forms.**PyNumberedDepartmentSelect**
A Select widget with a roman numbered list of Paraguayan departments as choices.

Peru (pe)

class pe.forms.**PEDNIField**
A form field that validates input as a DNI (Peruvian national identity) number.

class pe.forms.**PERUCField**
A form field that validates input as an RUC (Registro Unico de Contribuyentes) number. Valid RUC numbers have 11 digits.

class pe.forms.**PEDepartmentSelect**
A Select widget that uses a list of Peruvian Departments as its choices.

Poland (pl)

class pl.forms.**PLPESELField**
A form field that validates input as a Polish national identification number (PESEL).

New in version 1.4: *Please see the release notes*

class pl.forms.**PLNationalIDCardNumberField**
A form field that validates input as a Polish National ID Card number. The valid format is AAAXXXXXX, where A is letter (A-Z), X is digit and left-most digit is checksum digit. More information about checksum calculation algorithm see Polish identity card.

class pl.forms.**PLREGONField**
A form field that validates input as a Polish National Official Business Register Number (REGON), having either seven or nine digits. The checksum algorithm used for REGONs is documented at http://wipos.p.lodz.pl/zylla/ut/nip-rego.html.

class pl.forms.**PLPostalCodeField**
A form field that validates input as a Polish postal code. The valid format is XX-XXX, where X is a digit.

class pl.forms.**PLNIPField**
A form field that validates input as a Polish Tax Number (NIP). Valid formats are XXX-XXX-XX-XX, XXX-XX-XX-XXX or XXXXXXXXXX. The checksum algorithm used for NIPs is documented at http://wipos.p.lodz.pl/zylla/ut/nip-rego.html.

class pl.forms.**PLCountySelect**
A Select widget that uses a list of Polish administrative units as its choices.

class pl.forms.**PLProvinceSelect**
A Select widget that uses a list of Polish voivodeships (administrative provinces) as its choices.

Portugal (pt)

class pt.forms.**PTZipCodeField**
A form field that validates input as a Portuguese zip code.

class pt.forms.**PTPhoneNumberField**
A form field that validates input as a Portuguese phone number. Valid numbers have 9 digits (may include spaces) or start by 00 or + (international).

Romania (ro)

class ro.forms.**ROCIFField**
A form field that validates Romanian fiscal identification codes (CIF). The return value strips the leading RO, if given.

class ro.forms.**ROCNPField**
A form field that validates Romanian personal numeric codes (CNP).

class ro.forms.**ROCountyField**
A form field that validates its input as a Romanian county (judet) name or abbreviation. It normalizes the input to the standard vehicle registration abbreviation for the given county. This field will only accept names written with diacritics; consider using ROCountySelect as an alternative.

class ro.forms.**ROCountySelect**
A Select widget that uses a list of Romanian counties (judete) as its choices.

class ro.forms.**ROIBANField**
A form field that validates its input as a Romanian International Bank Account Number (IBAN). The valid format is ROXX-XXXX-XXXX-XXXX-XXXX-XXXX, with or without hyphens.

class ro.forms.**ROPhoneNumberField**
A form field that validates Romanian phone numbers, short special numbers excluded.

class ro.forms.**ROPostalCodeField**
A form field that validates Romanian postal codes.

Russia (ru)

New in version 1.4: *Please see the release notes*

class ru.forms.**RUPostalCodeField**
Russian Postal code field. The valid format is XXXXXX, where X is any digit and the first digit is not zero.

class ru.forms.**RUCountySelect**
A Select widget that uses a list of Russian Counties as its choices.

class ru.forms.**RURegionSelect**
A Select widget that uses a list of Russian Regions as its choices.

class ru.forms.**RUPassportNumberField**
Russian internal passport number. The valid format is XXXX XXXXXX, where X is any digit.

class ru.forms.**RUAlienPassportNumberField**
Russian alien's passport number. The valid format is XX XXXXXXX, where X is any digit.

Slovakia (sk)

class sk.forms.**SKPostalCodeField**
A form field that validates input as a Slovak postal code. Valid formats are XXXXX or XXX XX, where X is a digit.

class sk.forms.**SKDistrictSelect**
A Select widget that uses a list of Slovak districts as its choices.

class sk.forms.**SKRegionSelect**
A Select widget that uses a list of Slovak regions as its choices.

Slovenia (si)

class si.forms.**SIEMSOField**
A form field that validates input as Slovenian personal identification number and stores gender and birthday to self.info dictionary.

class si.forms.**SITaxNumberField**
A form field that validates input as a Slovenian tax number. Valid input is SIXXXXXXXX or XXXXXXXX.

class si.forms.**SIPhoneNumberField**
A form field that validates input as a Slovenian phone number. Phone number must contain at least local area code with optional country code.

class si.forms.**SIPostalCodeField**
A form field that provides a choice field of major Slovenian postal codes.

class si.forms.**SIPostalCodeSelect**
A Select widget that uses a list of major Slovenian postal codes as its choices.

South Africa (za)

class za.forms.**ZAIDField**
A form field that validates input as a South African ID number. Validation uses the Luhn checksum and a simplistic (i.e., not entirely accurate) check for birth date.

class za.forms.**ZAPostCodeField**
A form field that validates input as a South African postcode. Valid postcodes must have four digits.

Spain (es)

class es.forms.**ESIdentityCardNumberField**
A form field that validates input as a Spanish NIF/NIE/CIF (Fiscal Identification Number) code.

class es.forms.**ESCCCField**
A form field that validates input as a Spanish bank account number (Codigo Cuenta Cliente or CCC). A valid CCC number has the format EEEE-OOOO-CC-AAAAAAAAAA, where the E, O, C and A digits denote the entity, office, checksum and account, respectively. The first checksum digit validates the entity and office. The second checksum digit validates the account. It is also valid to use a space as a delimiter, or to use no delimiter.

class es.forms.**ESPhoneNumberField**
A form field that validates input as a Spanish phone number. Valid numbers have nine digits, the first of which is 6, 8 or 9.

class es.forms.**ESPostalCodeField**
A form field that validates input as a Spanish postal code. Valid codes have five digits, the first two being in the range 01 to 52, representing the province.

class es.forms.**ESProvinceSelect**
A Select widget that uses a list of Spanish provinces as its choices.

class es.forms.**ESRegionSelect**
A Select widget that uses a list of Spanish regions as its choices.

Sweden (se)

class se.forms.**SECountySelect**
A Select form widget that uses a list of the Swedish counties (län) as its choices.

The cleaned value is the official county code – see http://en.wikipedia.org/wiki/Counties_of_Sweden for a list.

class se.forms.**SEOrganisationNumber**
A form field that validates input as a Swedish organisation number (organisationsnummer).

It accepts the same input as SEPersonalIdentityField (for sole proprietorships (enskild firma). However, co-ordination numbers are not accepted.

It also accepts ordinary Swedish organisation numbers with the format NNNNNNNNNN.

The return value will be YYYYMMDDXXXX for sole proprietors, and NNNNNNNNNN for other organisations.

class se.forms.**SEPersonalIdentityNumber**
A form field that validates input as a Swedish personal identity number (personnummer).

The correct formats are YYYYMMDD-XXXX, YYYYMMDDXXXX, YYMMDD-XXXX, YYMMDDXXXX and YYMMDD+XXXX.

A + indicates that the person is older than 100 years, which will be taken into consideration when the date is validated.

The checksum will be calculated and checked. The birth date is checked to be a valid date.

By default, co-ordination numbers (samordningsnummer) will be accepted. To only allow real personal identity numbers, pass the keyword argument coordination_number=False to the constructor.

The cleaned value will always have the format YYYYMMDDXXXX.

class se.forms.**SEPostalCodeField**
A form field that validates input as a Swedish postal code (postnummer). Valid codes consist of five digits (XXXXX). The number can optionally be formatted with a space after the third digit (XXX XX).

The cleaned value will never contain the space.

Switzerland (ch)

class ch.forms.**CHIdentityCardNumberField**
A form field that validates input as a Swiss identity card number. A valid number must confirm to the X1234567<0 or 1234567890 format and have the correct checksums.

class ch.forms.**CHPhoneNumberField**
A form field that validates input as a Swiss phone number. The correct format is 0XX XXX XX XX. 0XX.XXX.XX.XX and 0XXXXXXXXX validate but are corrected to 0XX XXX XX XX.

class ch.forms.**CHZipCodeField**
A form field that validates input as a Swiss zip code. Valid codes consist of four digits.

class ch.forms.**CHStateSelect**
A Select widget that uses a list of Swiss states as its choices.

Turkey (tr)

class tr.forms.**TRZipCodeField**
A form field that validates input as a Turkish zip code. Valid codes consist of five digits.

class tr.forms.**TRPhoneNumberField**
A form field that validates input as a Turkish phone number. The correct format is 0xxx xxx xxxx. +90xxx xxx xxxx and inputs without spaces also validates. The result is normalized to xxx xxx xxxx format.

class tr.forms.**TRIdentificationNumberField**
A form field that validates input as a TR identification number. A valid number must satisfy the following:

- The number consist of 11 digits.
- The first digit cannot be 0.
- (sum(1st, 3rd, 5th, 7th, 9th)*7 - sum(2nd,4th,6th,8th)) % 10) must be equal to the 10th digit.
- (sum(1st to 10th) % 10) must be equal to the 11th digit.

class tr.forms.**TRProvinceSelect**
A select widget that uses a list of Turkish provinces as its choices.

United Kingdom (gb)

class gb.forms.**GBPostcodeField**
A form field that validates input as a UK postcode. The regular expression used is sourced from the schema for British Standard BS7666 address types at http://www.cabinetoffice.gov.uk/media/291293/bs7666-v2-0.xml.

class gb.forms.**GBCountySelect**
A Select widget that uses a list of UK counties/regions as its choices.

class gb.forms.**GBNationSelect**
A Select widget that uses a list of UK nations as its choices.

United States of America (us)

class us.forms.**USPhoneNumberField**
A form field that validates input as a U.S. phone number.

class us.forms.**USSocialSecurityNumberField**
A form field that validates input as a U.S. Social Security Number (SSN). A valid SSN must obey the following rules:

- Format of XXX-XX-XXXX
- No group of digits consisting entirely of zeroes
- Leading group of digits cannot be 666
- Number not in promotional block 987-65-4320 through 987-65-4329
- Number not one known to be invalid due to widespread promotional use or distribution (e.g., the Woolworth's number or the 1962 promotional number)

class us.forms.**USStateField**
A form field that validates input as a U.S. state name or abbreviation. It normalizes the input to the standard two-letter postal service abbreviation for the given state.

class us.forms.**USZipCodeField**
A form field that validates input as a U.S. ZIP code. Valid formats are XXXXX or XXXXX-XXXX.

class us.forms.**USStateSelect**
A form `Select` widget that uses a list of U.S. states/territories as its choices.

class us.forms.**USPSSelect**
A form `Select` widget that uses a list of U.S Postal Service state, territory and country abbreviations as its choices.

class us.models.**PhoneNumberField**
A `CharField` that checks that the value is a valid U.S.A.-style phone number (in the format `XXX-XXX-XXXX`).

class us.models.**USStateField**
A model field that forms represent as a `forms.USStateField` field and stores the two-letter U.S. state abbreviation in the database.

class us.models.**USPostalCodeField**
A model field that forms represent as a `forms.USPSSelect` field and stores the two-letter U.S Postal Service abbreviation in the database.

Additionally, a variety of choice tuples are provided in `django.contrib.localflavor.us.us_states`, allowing customized model and form fields, and form presentations, for subsets of U.S states, territories and U.S Postal Service abbreviations:

us.us_states.**CONTIGUOUS_STATES**
A tuple of choices of the postal abbreviations for the contiguous or "lower 48" states (i.e., all except Alaska and Hawaii), plus the District of Columbia.

us.us_states.**US_STATES**
A tuple of choices of the postal abbreviations for all 50 U.S. states, plus the District of Columbia.

us.us_states.**US_TERRITORIES**
A tuple of choices of the postal abbreviations for U.S territories: American Samoa, Guam, the Northern Mariana Islands, Puerto Rico and the U.S. Virgin Islands.

us.us_states.**ARMED_FORCES_STATES**
A tuple of choices of the postal abbreviations of the three U.S military postal "states": Armed Forces Americas, Armed Forces Europe and Armed Forces Pacific.

us.us_states.**COFA_STATES**
A tuple of choices of the postal abbreviations of the three independent nations which, under the Compact of Free

Association, are served by the U.S. Postal Service: the Federated States of Micronesia, the Marshall Islands and Palau.

`us.us_states.`**`OBSOLETE_STATES`**
A tuple of choices of obsolete U.S Postal Service state abbreviations: the former abbreviation for the Northern Mariana Islands, plus the Panama Canal Zone, the Philippines and the former Pacific trust territories.

`us.us_states.`**`STATE_CHOICES`**
A tuple of choices of all postal abbreviations corresponding to U.S states or territories, and the District of Columbia..

`us.us_states.`**`USPS_CHOICES`**
A tuple of choices of all postal abbreviations recognized by the U.S Postal Service (including all states and territories, the District of Columbia, armed forces "states" and independent nations serviced by USPS).

Uruguay (`uy`)

class `uy.forms.`**`UYCIField`**
A field that validates Uruguayan 'Cedula de identidad' (CI) numbers.

class `uy.forms.`**`UYDepartamentSelect`**
A `Select` widget that uses a list of Uruguayan departments as its choices.

6.4.12 django.contrib.markup

Django provides template filters that implement the following markup languages:

- `textile` – implements Textile – requires PyTextile
- `markdown` – implements Markdown – requires Python-markdown (>=2.1)
- `restructuredtext` – implements reST (reStructured Text) – requires doc-utils

In each case, the filter expects formatted markup as a string and returns a string representing the marked-up text. For example, the `textile` filter converts text that is marked-up in Textile format to HTML.

To activate these filters, add `'django.contrib.markup'` to your `INSTALLED_APPS` setting. Once you've done that, use `{% load markup %}` in a template, and you'll have access to these filters. For more documentation, read the source code in `django/contrib/markup/templatetags/markup.py`.

Warning: The output of markup filters is marked "safe" and will not be escaped when rendered in a template. Always be careful to sanitize your inputs and make sure you are not leaving yourself vulnerable to cross-site scripting or other types of attacks.

reStructured Text

When using the `restructuredtext` markup filter you can define a `RESTRUCTUREDTEXT_FILTER_SETTINGS` in your django settings to override the default writer settings. See the restructuredtext writer settings for details on what these settings are.

Warning: reStructured Text has features that allow raw HTML to be included, and that allow arbitrary files to be included. These can lead to XSS vulnerabilities and leaking of private information. It is your responsibility to check the features of this library and configure appropriately to avoid this. See the Deploying Docutils Securely documentation.

Markdown

The Python Markdown library supports options named "safe_mode" and "enable_attributes". Both relate to the security of the output. To enable both options in tandem, the markdown filter supports the "safe" argument:

```
{{ markdown_content_var|markdown:"safe" }}
```

Warning: Versions of the Python-Markdown library prior to 2.1 do not support the optional disabling of attributes. This is a security flaw. Therefore, `django.contrib.markup` has dropped support for versions of Python-Markdown < 2.1 in Django 1.5.

6.4.13 The messages framework

Django provides full support for cookie- and session-based messaging, for both anonymous and authenticated clients. The messages framework allows you to temporarily store messages in one request and retrieve them for display in a subsequent request (usually the next one). Every message is tagged with a specific `level` that determines its priority (e.g., `info`, `warning`, or `error`).

Enabling messages

Messages are implemented through a *middleware* class and corresponding *context processor*.

To enable message functionality, do the following:

- Edit the `MIDDLEWARE_CLASSES` setting and make sure it contains `'django.contrib.messages.middleware.MessageMiddleware'`.

  If you are using a *storage backend* that relies on *sessions* (the default), `'django.contrib.sessions.middleware.SessionMiddleware'` must be enabled and appear before `MessageMiddleware` in your `MIDDLEWARE_CLASSES`.

- Edit the `TEMPLATE_CONTEXT_PROCESSORS` setting and make sure it contains `'django.contrib.messages.context_processors.messages'`.
- Add `'django.contrib.messages'` to your `INSTALLED_APPS` setting

The default `settings.py` created by `django-admin.py startproject` has `MessageMiddleware` activated and the `django.contrib.messages` app installed. Also, the default value for `TEMPLATE_CONTEXT_PROCESSORS` contains `'django.contrib.messages.context_processors.messages'`.

If you don't want to use messages, you can remove the `MessageMiddleware` line from `MIDDLEWARE_CLASSES`, the `messages` context processor from `TEMPLATE_CONTEXT_PROCESSORS` and `'django.contrib.messages'` from your `INSTALLED_APPS`.

Configuring the message engine

Storage backends

The messages framework can use different backends to store temporary messages. If the default FallbackStorage isn't suitable to your needs, you can change which backend is being used by adding a MESSAGE_STORAGE to your settings, referencing the module and class of the storage class. For example:

```
MESSAGE_STORAGE = 'django.contrib.messages.storage.cookie.CookieStorage'
```

The value should be the full path of the desired storage class.

Three storage classes are available:

'django.contrib.messages.storage.session.SessionStorage' This class stores all messages inside of the request's session. It requires Django's `contrib.sessions` application.

'django.contrib.messages.storage.cookie.CookieStorage' This class stores the message data in a cookie (signed with a secret hash to prevent manipulation) to persist notifications across requests. Old messages are dropped if the cookie data size would exceed 4096 bytes.

'django.contrib.messages.storage.fallback.FallbackStorage' This is the default storage class.

This class first uses CookieStorage for all messages, falling back to using SessionStorage for the messages that could not fit in a single cookie.

Since it is uses SessionStorage, it also requires Django's `contrib.sessions` application.

To write your own storage class, subclass the `BaseStorage` class in `django.contrib.messages.storage.base` and implement the `_get` and `_store` methods.

Message levels

The messages framework is based on a configurable level architecture similar to that of the Python logging module. Message levels allow you to group messages by type so they can be filtered or displayed differently in views and templates.

The built-in levels (which can be imported from `django.contrib.messages` directly) are:

Constant	Purpose
`DEBUG`	Development-related messages that will be ignored (or removed) in a production deployment
`INFO`	Informational messages for the user
`SUCCESS`	An action was successful, e.g. "Your profile was updated successfully"
`WARNING`	A failure did not occur but may be imminent
`ERROR`	An action was **not** successful or some other failure occurred

The MESSAGE_LEVEL setting can be used to change the minimum recorded level (or it can be changed per request). Attempts to add messages of a level less than this will be ignored.

Message tags

Message tags are a string representation of the message level plus any extra tags that were added directly in the view (see Adding extra message tags below for more details). Tags are stored in a string and are separated by spaces. Typically, message tags are used as CSS classes to customize message style based on message type. By default, each level has a single tag that's a lowercase version of its own constant:

Level Constant	Tag
`DEBUG`	`debug`
`INFO`	`info`
`SUCCESS`	`success`
`WARNING`	`warning`
`ERROR`	`error`

To change the default tags for a message level (either built-in or custom), set the MESSAGE_TAGS setting to a dictionary containing the levels you wish to change. As this extends the default tags, you only need to provide tags for the levels you wish to override:

```
from django.contrib.messages import constants as messages
MESSAGE_TAGS = {
    messages.INFO: '',
    50: 'critical',
}
```

Using messages in views and templates

Adding a message

To add a message, call:

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Hello world.')
```

Some shortcut methods provide a standard way to add messages with commonly used tags (which are usually represented as HTML classes for the message):

```
messages.debug(request, '%s SQL statements were executed.' % count)
messages.info(request, 'Three credits remain in your account.')
messages.success(request, 'Profile details updated.')
messages.warning(request, 'Your account expires in three days.')
messages.error(request, 'Document deleted.')
```

Displaying messages

In your template, use something like:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

If you're using the context processor, your template should be rendered with a `RequestContext`. Otherwise, ensure `messages` is available to the template context.

Even if you know there is only just one message, you should still iterate over the `messages` sequence, because otherwise the message storage will not be cleared for the next request.

Creating custom message levels

Messages levels are nothing more than integers, so you can define your own level constants and use them to create more customized user feedback, e.g.:

```
CRITICAL = 50

def my_view(request):
    messages.add_message(request, CRITICAL, 'A serious error occurred.')
```

When creating custom message levels you should be careful to avoid overloading existing levels. The values for the built-in levels are:

Level Constant	Value
DEBUG	10
INFO	20
SUCCESS	25
WARNING	30
ERROR	40

If you need to identify the custom levels in your HTML or CSS, you need to provide a mapping via the MESSAGE_TAGS setting.

Note: If you are creating a reusable application, it is recommended to use only the built-in message levels and not rely on any custom levels.

Changing the minimum recorded level per-request

The minimum recorded level can be set per request via the `set_level` method:

```
from django.contrib import messages

# Change the messages level to ensure the debug message is added.
messages.set_level(request, messages.DEBUG)
messages.debug(request, 'Test message...')

# In another request, record only messages with a level of WARNING and higher
messages.set_level(request, messages.WARNING)
messages.success(request, 'Your profile was updated.') # ignored
messages.warning(request, 'Your account is about to expire.') # recorded

# Set the messages level back to default.
messages.set_level(request, None)
```

Similarly, the current effective level can be retrieved with `get_level`:

```
from django.contrib import messages
current_level = messages.get_level(request)
```

For more information on how the minimum recorded level functions, see Message levels above.

Adding extra message tags

For more direct control over message tags, you can optionally provide a string containing extra tags to any of the add methods:

```
messages.add_message(request, messages.INFO, 'Over 9000!',
                     extra_tags='dragonball')
messages.error(request, 'Email box full', extra_tags='email')
```

Extra tags are added before the default tag for that level and are space separated.

Failing silently when the message framework is disabled

If you're writing a reusable app (or other piece of code) and want to include messaging functionality, but don't want to require your users to enable it if they don't want to, you may pass an additional keyword argument

`fail_silently=True` to any of the `add_message` family of methods. For example:

```
messages.add_message(request, messages.SUCCESS, 'Profile details updated.',
                     fail_silently=True)
messages.info(request, 'Hello world.', fail_silently=True)
```

Note: Setting `fail_silently=True` only hides the `MessageFailure` that would otherwise occur when the messages framework disabled and one attempts to use one of the `add_message` family of methods. It does not hide failures that may occur for other reasons.

Expiration of messages

The messages are marked to be cleared when the storage instance is iterated (and cleared when the response is processed).

To avoid the messages being cleared, you can set the messages storage to `False` after iterating:

```
storage = messages.get_messages(request)
for message in storage:
    do_something_with(message)
storage.used = False
```

Behavior of parallel requests

Due to the way cookies (and hence sessions) work, **the behavior of any backends that make use of cookies or sessions is undefined when the same client makes multiple requests that set or get messages in parallel**. For example, if a client initiates a request that creates a message in one window (or tab) and then another that fetches any uniterated messages in another window, before the first window redirects, the message may appear in the second window instead of the first window where it may be expected.

In short, when multiple simultaneous requests from the same client are involved, messages are not guaranteed to be delivered to the same window that created them nor, in some cases, at all. Note that this is typically not a problem in most applications and will become a non-issue in HTML5, where each window/tab will have its own browsing context.

Settings

A few *Django settings* give you control over message behavior:

MESSAGE_LEVEL

Default: `messages.INFO`

This sets the minimum message that will be saved in the message storage. See Message levels above for more details.

Important

If you override `MESSAGE_LEVEL` in your settings file and rely on any of the built-in constants, you must import the constants module directly to avoid the potential for circular imports, e.g.:

```
from django.contrib.messages import constants as message_constants
MESSAGE_LEVEL = message_constants.DEBUG
```

If desired, you may specify the numeric values for the constants directly according to the values in the above *constants table*.

MESSAGE_STORAGE

Default: `'django.contrib.messages.storage.user_messages.FallbackStorage'`

Controls where Django stores message data. Valid values are:

- `'django.contrib.messages.storage.fallback.FallbackStorage'`
- `'django.contrib.messages.storage.session.SessionStorage'`
- `'django.contrib.messages.storage.cookie.CookieStorage'`

See Storage backends for more details.

MESSAGE_TAGS

Default:

```
{messages.DEBUG: 'debug',
messages.INFO: 'info',
messages.SUCCESS: 'success',
messages.WARNING: 'warning',
messages.ERROR: 'error',}
```

This sets the mapping of message level to message tag, which is typically rendered as a CSS class in HTML. If you specify a value, it will extend the default. This means you only have to specify those values which you need to override. See Displaying messages above for more details.

Important

If you override `MESSAGE_TAGS` in your settings file and rely on any of the built-in constants, you must import the `constants` module directly to avoid the potential for circular imports, e.g.:

```
from django.contrib.messages import constants as message_constants
MESSAGE_TAGS = {message_constants.INFO: ''}
```

If desired, you may specify the numeric values for the constants directly according to the values in the above *constants table*.

SESSION_COOKIE_DOMAIN

Default: `None`

The storage backends that use cookies – `CookieStorage` and `FallbackStorage` – use the value of `SESSION_COOKIE_DOMAIN` in setting their cookies. See the *settings documentation* for more information on how this works and why you might need to set it.

6.4.14 The redirects app

Django comes with an optional redirects application. It lets you store simple redirects in a database and handles the redirecting for you.

Installation

To install the redirects app, follow these steps:

1. Add `'django.contrib.redirects'` to your `INSTALLED_APPS` setting.
2. Add `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` to your `MIDDLEWARE_CLASSES` setting.
3. Run the command `manage.py syncdb`.

How it works

`manage.py syncdb` creates a `django_redirect` table in your database. This is a simple lookup table with `site_id`, `old_path` and `new_path` fields.

The `RedirectFallbackMiddleware` does all of the work. Each time any Django application raises a 404 error, this middleware checks the redirects database for the requested URL as a last resort. Specifically, it checks for a redirect with the given `old_path` with a site ID that corresponds to the `SITE_ID` setting.

- If it finds a match, and `new_path` is not empty, it redirects to `new_path`.
- If it finds a match, and `new_path` is empty, it sends a 410 ("Gone") HTTP header and empty (content-less) response.
- If it doesn't find a match, the request continues to be processed as usual.

The middleware only gets activated for 404s – not for 500s or responses of any other status code.

Note that the order of `MIDDLEWARE_CLASSES` matters. Generally, you can put `RedirectFallbackMiddleware` at the end of the list, because it's a last resort.

For more on middleware, read the *middleware docs*.

How to add, change and delete redirects

Via the admin interface

If you've activated the automatic Django admin interface, you should see a "Redirects" section on the admin index page. Edit redirects as you edit any other object in the system.

Via the Python API

class `models.`**`Redirect`**

Redirects are represented by a standard *Django model*, which lives in django/contrib/redirects/models.py. You can access redirect objects via the *Django database API*.

6.4.15 The sitemap framework

Django comes with a high-level sitemap-generating framework that makes creating sitemap XML files easy.

Overview

A sitemap is an XML file on your Web site that tells search-engine indexers how frequently your pages change and how "important" certain pages are in relation to other pages on your site. This information helps search engines index your site.

The Django sitemap framework automates the creation of this XML file by letting you express this information in Python code.

It works much like Django's *syndication framework*. To create a sitemap, just write a `Sitemap` class and point to it in your *URLconf*.

Installation

To install the sitemap app, follow these steps:

1. Add `'django.contrib.sitemaps'` to your `INSTALLED_APPS` setting.
2. Make sure `'django.template.loaders.app_directories.Loader'` is in your `TEMPLATE_LOADERS` setting. It's in there by default, so you'll only need to change this if you've changed that setting.
3. Make sure you've installed the `sites framework`.

(Note: The sitemap application doesn't install any database tables. The only reason it needs to go into `INSTALLED_APPS` is so that the `Loader()` template loader can find the default templates.)

Initialization

To activate sitemap generation on your Django site, add this line to your *URLconf*:

```
(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})
```

This tells Django to build a sitemap when a client accesses `/sitemap.xml`.

The name of the sitemap file is not important, but the location is. Search engines will only index links in your sitemap for the current URL level and below. For instance, if `sitemap.xml` lives in your root directory, it may reference any URL in your site. However, if your sitemap lives at `/content/sitemap.xml`, it may only reference URLs that begin with `/content/`.

The sitemap view takes an extra, required argument: `{'sitemaps': sitemaps}`. `sitemaps` should be a dictionary that maps a short section label (e.g., `blog` or `news`) to its `Sitemap` class (e.g., `BlogSitemap` or `NewsSitemap`). It may also map to an *instance* of a `Sitemap` class (e.g., `BlogSitemap(some_var)`).

Sitemap classes

A `Sitemap` class is a simple Python class that represents a "section" of entries in your sitemap. For example, one `Sitemap` class could represent all the entries of your Weblog, while another could represent all of the events in your events calendar.

In the simplest case, all these sections get lumped together into one `sitemap.xml`, but it's also possible to use the framework to generate a sitemap index that references individual sitemap files, one per section. (See Creating a sitemap index below.)

`Sitemap` classes must subclass `django.contrib.sitemaps.Sitemap`. They can live anywhere in your codebase.

A simple example

Let's assume you have a blog system, with an `Entry` model, and you want your sitemap to include all the links to your individual blog entries. Here's how your sitemap class might look:

```
from django.contrib.sitemaps import Sitemap
from blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

Note:

- `changefreq` and `priority` are class attributes corresponding to `<changefreq>` and `<priority>` elements, respectively. They can be made callable as functions, as `lastmod` was in the example.
- `items()` is simply a method that returns a list of objects. The objects returned will get passed to any callable methods corresponding to a sitemap property (`location`, `lastmod`, `changefreq`, and `priority`).
- `lastmod` should return a Python `datetime` object.
- There is no `location` method in this example, but you can provide it in order to specify the URL for your object. By default, `location()` calls `get_absolute_url()` on each object and returns the result.

Sitemap class reference

class `Sitemap`

A `Sitemap` class can define the following methods/attributes:

`items`

Required. A method that returns a list of objects. The framework doesn't care what *type* of objects they are; all that matters is that these objects get passed to the `location()`, `lastmod()`, `changefreq()` and `priority()` methods.

`location`

Optional. Either a method or attribute.

If it's a method, it should return the absolute path for a given object as returned by `items()`.

If it's an attribute, its value should be a string representing an absolute path to use for *every* object returned by `items()`.

In both cases, "absolute path" means a URL that doesn't include the protocol or domain. Examples:

- Good: `'/foo/bar/'`
- Bad: `'example.com/foo/bar/'`
- Bad: `'http://example.com/foo/bar/'`

If `location` isn't provided, the framework will call the `get_absolute_url()` method on each object as returned by `items()`.

To specify a protocol other than `'http'`, use `protocol`.

lastmod

Optional. Either a method or attribute.

If it's a method, it should take one argument – an object as returned by `items()` – and return that object's last-modified date/time, as a Python `datetime.datetime` object.

If it's an attribute, its value should be a Python `datetime.datetime` object representing the last-modified date/time for *every* object returned by `items()`.

changefreq

Optional. Either a method or attribute.

If it's a method, it should take one argument – an object as returned by `items()` – and return that object's change frequency, as a Python string.

If it's an attribute, its value should be a string representing the change frequency of *every* object returned by `items()`.

Possible values for `changefreq`, whether you use a method or attribute, are:

- `'always'`
- `'hourly'`
- `'daily'`
- `'weekly'`
- `'monthly'`
- `'yearly'`
- `'never'`

priority

Optional. Either a method or attribute.

If it's a method, it should take one argument – an object as returned by `items()` – and return that object's priority, as either a string or float.

If it's an attribute, its value should be either a string or float representing the priority of *every* object returned by `items()`.

Example values for `priority`: `0.4`, `1.0`. The default priority of a page is `0.5`. See the sitemaps.org documentation for more.

protocol

New in version 1.4: *Please see the release notes* **Optional.**

This attribute defines the protocol (`'http'` or `'https'`) of the URLs in the sitemap. If it isn't set, the protocol with which the sitemap was requested is used. If the sitemap is built outside the context of a request, the default is `'http'`.

Shortcuts

The sitemap framework provides a couple convenience classes for common cases:

class FlatPageSitemap

The `django.contrib.sitemaps.FlatPageSitemap` class looks at all publicly visible `flatpages` defined for the current `SITE_ID` (see the `sites` documentation) and creates an entry in the sitemap. These entries include only the `location` attribute – not `lastmod`, `changefreq` or `priority`.

class GenericSitemap

The `django.contrib.sitemaps.GenericSitemap` class allows you to create a sitemap by passing it a dictionary which has to contain at least a `queryset` entry. This queryset will be used to generate the items of the sitemap. It may also have a `date_field` entry that specifies a date field for objects retrieved from the `queryset`. This will be used for the `lastmod` attribute in the generated sitemap. You may also pass `priority` and `changefreq` keyword arguments to the `GenericSitemap` constructor to specify these attributes for all URLs.

Example

Here's an example of a *URLconf* using both:

```
from django.conf.urls import patterns, url, include
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

sitemaps = {
    'flatpages': FlatPageSitemap,
    'blog': GenericSitemap(info_dict, priority=0.6),
}

urlpatterns = patterns('',
    # some generic view using info_dict
    # ...

    # the sitemap
    (r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps})
)
```

Creating a sitemap index

The sitemap framework also has the ability to create a sitemap index that references individual sitemap files, one per each section defined in your `sitemaps` dictionary. The only differences in usage are:

- You use two views in your URLconf: `django.contrib.sitemaps.views.index()` and `django.contrib.sitemaps.views.sitemap()`.
- The `django.contrib.sitemaps.views.sitemap()` view should take a `section` keyword argument.

Here's what the relevant URLconf lines would look like for the example above:

```
urlpatterns = patterns('django.contrib.sitemaps.views',
    (r'^sitemap\.xml$', 'index', {'sitemaps': sitemaps}),
    (r'^sitemap-(?P<section>.+)\.xml$', 'sitemap', {'sitemaps': sitemaps}),
)
```

This will automatically generate a `sitemap.xml` file that references both `sitemap-flatpages.xml` and `sitemap-blog.xml`. The `Sitemap` classes and the `sitemaps` dict don't change at all.

You should create an index file if one of your sitemaps has more than 50,000 URLs. In this case, Django will automatically paginate the sitemap, and the index will reflect that. New in version 1.4: *Please see the release notes* If

you're not using the vanilla sitemap view – for example, if it's wrapped with a caching decorator – you must name your sitemap view and pass `sitemap_url_name` to the index view:

```
from django.contrib.sitemaps import views as sitemaps_views
from django.views.decorators.cache import cache_page

urlpatterns = patterns('',
    url(r'^sitemap.xml$',
        cache_page(86400)(sitemaps_views.index),
        {'sitemaps': sitemaps, 'sitemap_url_name': 'sitemaps'}),
    url(r'^sitemap-(?P<section>.+)\.xml$',
        cache_page(86400)(sitemaps_views.sitemap),
        {'sitemaps': sitemaps}, name='sitemaps'),
)
```

Template customization

New in version 1.3: *Please see the release notes* If you wish to use a different template for each sitemap or sitemap index available on your site, you may specify it by passing a `template_name` parameter to the `sitemap` and `index` views via the URLconf:

```
urlpatterns = patterns('django.contrib.sitemaps.views',
    (r'^custom-sitemap\.xml$', 'index', {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
    (r'^custom-sitemap-(?P<section>.+)\.xml$', 'sitemap', {
        'sitemaps': sitemaps,
        'template_name': 'custom_sitemap.html'
    }),
)
```

Changed in version 1.4: In addition, these views also return `TemplateResponse` instances which allow you to easily customize the response data before rendering. For more details, see the *TemplateResponse documentation*.

Context variables

When customizing the templates for the `index()` and `sitemaps()` views, you can rely on the following context variables.

Index

The variable `sitemaps` is a list of absolute URLs to each of the sitemaps.

Sitemap

The variable `urlset` is a list of URLs that should appear in the sitemap. Each URL exposes attributes as defined in the `Sitemap` class:

- `changefreq`
- `item`
- `lastmod`

- location
- priority

New in version 1.4: *Please see the release notes* The item attribute has been added for each URL to allow more flexible customization of the templates, such as Google news sitemaps. Assuming Sitemap's items() would return a list of items with publication_data and a tags field something like this would generate a Google News compatible sitemap:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset
  xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
  xmlns:news="http://www.google.com/schemas/sitemap-news/0.9">
{% spaceless %}
{% for url in urlset %}
  <url>
    <loc>{{ url.location }}</loc>
    {% if url.lastmod %}<lastmod>{{ url.lastmod|date:"Y-m-d" }}</lastmod>{% endif %}
    {% if url.changefreq %}<changefreq>{{ url.changefreq }}</changefreq>{% endif %}
    {% if url.priority %}<priority>{{ url.priority }}</priority>{% endif %}
    <news:news>
      {% if url.item.publication_date %}<news:publication_date>{{ url.item.publication_date|date:"Y-m
      {% if url.item.tags %}<news:keywords>{{ url.item.tags }}</news:keywords>{% endif %}
    </news:news>
  </url>
{% endfor %}
{% endspaceless %}
</urlset>
```

Pinging Google

You may want to "ping" Google when your sitemap changes, to let it know to reindex your site. The sitemaps framework provides a function to do just that: django.contrib.sitemaps.ping_google().

ping_google()

ping_google() takes an optional argument, sitemap_url, which should be the absolute path to your site's sitemap (e.g., '/sitemap.xml'). If this argument isn't provided, ping_google() will attempt to figure out your sitemap by performing a reverse looking in your URLconf.

ping_google() raises the exception django.contrib.sitemaps.SitemapNotFound if it cannot determine your sitemap URL.

Register with Google first!

The ping_google() command only works if you have registered your site with Google Webmaster Tools.

One useful way to call ping_google() is from a model's save() method:

```
from django.contrib.sitemaps import ping_google

 class Entry(models.Model):
     # ...
     def save(self, force_insert=False, force_update=False):
         super(Entry, self).save(force_insert, force_update)
         try:
             ping_google()
         except Exception:
             # Bare 'except' because we could get a variety
```

```
            # of HTTP-related exceptions.
            pass
```

A more efficient solution, however, would be to call `ping_google()` from a cron script, or some other scheduled task. The function makes an HTTP request to Google's servers, so you may not want to introduce that network overhead each time you call `save()`.

Pinging Google via *manage.py*

`django-admin.py ping_google`

Once the sitemaps application is added to your project, you may also ping Google using the `ping_google` management command:

```
python manage.py ping_google [/sitemap.xml]
```

6.4.16 The "sites" framework

Django comes with an optional "sites" framework. It's a hook for associating objects and functionality to particular Web sites, and it's a holding place for the domain names and "verbose" names of your Django-powered sites.

Use it if your single Django installation powers more than one site and you need to differentiate between those sites in some way.

The whole sites framework is based on a simple model:

class `Site`

A model for storing the `domain` and `name` attributes of a Web site. The `SITE_ID` setting specifies the database ID of the `Site` object associated with that particular settings file.

`domain`

The domain name associated with the Web site.

`name`

A human-readable "verbose" name for the Web site.

How you use this is up to you, but Django uses it in a couple of ways automatically via simple conventions.

Example usage

Why would you use sites? It's best explained through examples.

Associating content with multiple sites

The Django-powered sites LJWorld.com and Lawrence.com are operated by the same news organization – the Lawrence Journal-World newspaper in Lawrence, Kansas. LJWorld.com focuses on news, while Lawrence.com focuses on local entertainment. But sometimes editors want to publish an article on *both* sites.

The brain-dead way of solving the problem would be to require site producers to publish the same story twice: once for LJWorld.com and again for Lawrence.com. But that's inefficient for site producers, and it's redundant to store multiple copies of the same story in the database.

The better solution is simple: Both sites use the same article database, and an article is associated with one or more sites. In Django model terminology, that's represented by a `ManyToManyField` in the `Article` model:

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    sites = models.ManyToManyField(Site)
```

This accomplishes several things quite nicely:

- It lets the site producers edit all content – on both sites – in a single interface (the Django admin).
- It means the same story doesn't have to be published twice in the database; it only has a single record in the database.
- It lets the site developers use the same Django view code for both sites. The view code that displays a given story just checks to make sure the requested story is on the current site. It looks something like this:

```
from django.conf import settings

def article_detail(request, article_id):
    try:
        a = Article.objects.get(id=article_id, sites__id__exact=settings.SITE_ID)
    except Article.DoesNotExist:
        raise Http404
    # ...
```

Associating content with a single site

Similarly, you can associate a model to the `Site` model in a many-to-one relationship, using `ForeignKey`.

For example, if an article is only allowed on a single site, you'd use a model like this:

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    site = models.ForeignKey(Site)
```

This has the same benefits as described in the last section.

Hooking into the current site from views

You can use the sites framework in your Django views to do particular things based on the site in which the view is being called. For example:

```
from django.conf import settings

def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
    else:
        # Do something else.
```

Of course, it's ugly to hard-code the site IDs like that. This sort of hard-coding is best for hackish fixes that you need done quickly. A slightly cleaner way of accomplishing the same thing is to check the current site's domain:

```
from django.conf import settings
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get(id=settings.SITE_ID)
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

The idiom of retrieving the `Site` object for the value of `settings.SITE_ID` is quite common, so the `Site` model's manager has a `get_current()` method. This example is equivalent to the previous one:

```
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.
```

Changed in version 1.3: *Please see the release notes* For code which relies on getting the current domain but cannot be certain that the sites framework will be installed for any given project, there is a utility function `get_current_site()` that takes a request object as an argument and returns either a Site instance (if the sites framework is installed) or a RequestSite instance (if it is not). This allows loose coupling with the sites framework and provides a usable fallback for cases where it is not installed. New in version 1.3: *Please see the release notes*

get_current_site(*request*)
Checks if contrib.sites is installed and returns either the current `Site` object or a `RequestSite` object based on the request.

Getting the current domain for display

LJWorld.com and Lawrence.com both have email alert functionality, which lets readers sign up to get notifications when news happens. It's pretty basic: A reader signs up on a Web form, and he immediately gets an email saying, "Thanks for your subscription."

It'd be inefficient and redundant to implement this signup-processing code twice, so the sites use the same code behind the scenes. But the "thank you for signing up" notice needs to be different for each site. By using `Site` objects, we can abstract the "thank you" notice to use the values of the current site's `name` and `domain`.

Here's an example of what the form-handling view looks like:

```
from django.contrib.sites.models import Site
from django.core.mail import send_mail

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    current_site = Site.objects.get_current()
    send_mail('Thanks for subscribing to %s alerts' % current_site.name,
        'Thanks for your subscription. We appreciate it.\n\n-The %s team.' % current_site.name,
        'editor@%s' % current_site.domain,
```

```
        [user.email])

    # ...
```

On Lawrence.com, this email has the subject line "Thanks for subscribing to lawrence.com alerts." On LJWorld.com, the email has the subject "Thanks for subscribing to LJWorld.com alerts." Same goes for the email's message body.

Note that an even more flexible (but more heavyweight) way of doing this would be to use Django's template system. Assuming Lawrence.com and LJWorld.com have different template directories (`TEMPLATE_DIRS`), you could simply farm out to the template system like so:

```
from django.core.mail import send_mail
from django.template import loader, Context

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    subject = loader.get_template('alerts/subject.txt').render(Context({}))
    message = loader.get_template('alerts/message.txt').render(Context({}))
    send_mail(subject, message, 'editor@ljworld.com', [user.email])

    # ...
```

In this case, you'd have to create `subject.txt` and `message.txt` template files for both the LJWorld.com and Lawrence.com template directories. That gives you more flexibility, but it's also more complex.

It's a good idea to exploit the `Site` objects as much as possible, to remove unneeded complexity and redundancy.

Getting the current domain for full URLs

Django's `get_absolute_url()` convention is nice for getting your objects' URL without the domain name, but in some cases you might want to display the full URL – with `http://` and the domain and everything – for an object. To do this, you can use the sites framework. A simple example:

```
>>> from django.contrib.sites.models import Site
>>> obj = MyModel.objects.get(id=3)
>>> obj.get_absolute_url()
'/mymodel/objects/3/'
>>> Site.objects.get_current().domain
'example.com'
>>> 'http://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'http://example.com/mymodel/objects/3/'
```

Caching the current `Site` object

As the current site is stored in the database, each call to `Site.objects.get_current()` could result in a database query. But Django is a little cleverer than that: on the first request, the current site is cached, and any subsequent call returns the cached data instead of hitting the database.

If for any reason you want to force a database query, you can tell Django to clear the cache using `Site.objects.clear_cache()`:

```
# First call; current site fetched from database.
current_site = Site.objects.get_current()
# ...
```

```
# Second call; current site fetched from cache.
current_site = Site.objects.get_current()
# ...

# Force a database query for the third call.
Site.objects.clear_cache()
current_site = Site.objects.get_current()
```

The `CurrentSiteManager`

class `CurrentSiteManager`

If `Site` plays a key role in your application, consider using the helpful `CurrentSiteManager` in your model(s). It's a model *manager* that automatically filters its queries to include only objects associated with the current `Site`.

Use `CurrentSiteManager` by adding it to your model explicitly. For example:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager()
```

With this model, `Photo.objects.all()` will return all `Photo` objects in the database, but `Photo.on_site.all()` will return only the `Photo` objects associated with the current site, according to the `SITE_ID` setting.

Put another way, these two statements are equivalent:

```
Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()
```

How did `CurrentSiteManager` know which field of `Photo` was the `Site`? By default, `CurrentSiteManager` looks for a either a `ForeignKey` called `site` or a `ManyToManyField` called `sites` to filter on. If you use a field named something other than `site` or `sites` to identify which `Site` objects your object is related to, then you need to explicitly pass the custom field name as a parameter to `CurrentSiteManager` on your model. The following model, which has a field called `publish_on`, demonstrates this:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

If you attempt to use `CurrentSiteManager` and pass a field name that doesn't exist, Django will raise a `ValueError`.

Finally, note that you'll probably want to keep a normal (non-site-specific) `Manager` on your model, even if you use `CurrentSiteManager`. As explained in the *manager documentation*, if you define a manager manually, then Django won't create the automatic `objects = models.Manager()` manager for you. Also note that certain parts of Django – namely, the Django admin site and generic views – use whichever manager is defined *first* in the model, so if you want your admin site to have access to all objects (not just site-specific ones), put `objects = models.Manager()` in your model, before you define `CurrentSiteManager`.

How Django uses the sites framework

Although it's not required that you use the sites framework, it's strongly encouraged, because Django takes advantage of it in a few places. Even if your Django installation is powering only a single site, you should take the two seconds to create the site object with your `domain` and `name`, and point to its ID in your `SITE_ID` setting.

Here's how Django uses the sites framework:

- In the `redirects framework`, each redirect object is associated with a particular site. When Django searches for a redirect, it takes into account the current `SITE_ID`.
- In the comments framework, each comment is associated with a particular site. When a comment is posted, its `Site` is set to the current `SITE_ID`, and when comments are listed via the appropriate template tag, only the comments for the current site are displayed.
- In the `flatpages framework`, each flatpage is associated with a particular site. When a flatpage is created, you specify its `Site`, and the `FlatpageFallbackMiddleware` checks the current `SITE_ID` in retrieving flatpages to display.
- In the `syndication framework`, the templates for `title` and `description` automatically have access to a variable `{{ site }}`, which is the `Site` object representing the current site. Also, the hook for providing item URLs will use the `domain` from the current `Site` object if you don't specify a fully-qualified domain.
- In the `authentication framework`, the `django.contrib.auth.views.login()` view passes the current `Site` name to the template as `{{ site_name }}`.
- The shortcut view (`django.views.defaults.shortcut`) uses the domain of the current `Site` object when calculating an object's URL.
- In the admin framework, the "view on site" link uses the current `Site` to work out the domain for the site that it will redirect to.

`RequestSite` objects

Some *django.contrib* applications take advantage of the sites framework but are architected in a way that doesn't *require* the sites framework to be installed in your database. (Some people don't want to, or just aren't *able* to install the extra database table that the sites framework requires.) For those cases, the framework provides a `RequestSite` class, which can be used as a fallback when the database-backed sites framework is not available.

class `RequestSite`

A class that shares the primary interface of `Site` (i.e., it has `domain` and `name` attributes) but gets its data from a Django `HttpRequest` object rather than from a database.

The `save()` and `delete()` methods raise `NotImplementedError`.

`__init__`(*request*)

Sets the `name` and `domain` attributes to the value of `get_host()`.

A `RequestSite` object has a similar interface to a normal `Site` object, except its `__init__()` method takes an `HttpRequest` object. It's able to deduce the `domain` and `name` by looking at the request's domain. It has `save()` and `delete()` methods to match the interface of `Site`, but the methods raise `NotImplementedError`.

6.4.17 The staticfiles app

New in version 1.3: *Please see the release notes* `django.contrib.staticfiles` collects static files from each of your applications (and any other places you specify) into a single location that can easily be served in production.

See Also:

For an introduction to the static files app and some usage examples, see *Managing static files*.

Settings

Note: The following settings control the behavior of the staticfiles app.

STATICFILES_DIRS

Default: `[]`

This setting defines the additional locations the staticfiles app will traverse if the `FileSystemFinder` finder is enabled, e.g. if you use the `collectstatic` or `findstatic` management command or use the static file serving view.

This should be set to a list or tuple of strings that contain full paths to your additional files directory(ies) e.g.:

```
STATICFILES_DIRS = (
    "/home/special.polls.com/polls/static",
    "/home/polls.com/polls/static",
    "/opt/webfiles/common",
)
```

Prefixes (optional) In case you want to refer to files in one of the locations with an additional namespace, you can **optionally** provide a prefix as `(prefix, path)` tuples, e.g.:

```
STATICFILES_DIRS = (
    # ...
    ("downloads", "/opt/webfiles/stats"),
)
```

Example:

Assuming you have `STATIC_URL` set `'/static/'`, the `collectstatic` management command would collect the "stats" files in a `'downloads'` subdirectory of `STATIC_ROOT`.

This would allow you to refer to the local file `'/opt/webfiles/stats/polls_20101022.tar.gz'` with `'/static/downloads/polls_20101022.tar.gz'` in your templates, e.g.:

```
<a href="{{ STATIC_URL }}downloads/polls_20101022.tar.gz">
```

STATICFILES_STORAGE

Default: `'django.contrib.staticfiles.storage.StaticFilesStorage'`

The file storage engine to use when collecting static files with the `collectstatic` management command. New in version 1.4: *Please see the release notes* A ready-to-use instance of the storage backend defined in this setting can be found at `django.contrib.staticfiles.storage.staticfiles_storage`.

For an example, see *Serving static files from a cloud service or CDN*.

STATICFILES_FINDERS

Default:

```
("django.contrib.staticfiles.finders.FileSystemFinder",
 "django.contrib.staticfiles.finders.AppDirectoriesFinder")
```

The list of finder backends that know how to find static files in various locations.

The default will find files stored in the `STATICFILES_DIRS` setting (using `django.contrib.staticfiles.finders.FileSystemFinder`) and in a `static` subdirectory of each app (using `django.contrib.staticfiles.finders.AppDirectoriesFinder`)

One finder is disabled by default: `django.contrib.staticfiles.finders.DefaultStorageFinder`. If added to your `STATICFILES_FINDERS` setting, it will look for static files in the default file storage as defined by the `DEFAULT_FILE_STORAGE` setting.

Note: When using the `AppDirectoriesFinder` finder, make sure your apps can be found by staticfiles. Simply add the app to the `INSTALLED_APPS` setting of your site.

Static file finders are currently considered a private interface, and this interface is thus undocumented.

Management Commands

`django.contrib.staticfiles` exposes three management commands.

collectstatic

`django-admin.py collectstatic`

Collects the static files into `STATIC_ROOT`.

Duplicate file names are by default resolved in a similar way to how template resolution works: the file that is first found in one of the specified locations will be used. If you're confused, the `findstatic` command can help show you which files are found.

Files are searched by using the `enabled finders`. The default is to look in all locations defined in `STATICFILES_DIRS` and in the `'static'` directory of apps specified by the `INSTALLED_APPS` setting. New in version 1.4: *Please see the release notes* The `collectstatic` management command calls the `post_process()` method of the `STATICFILES_STORAGE` after each run and passes a list of paths that have been found by the management command. It also receives all command line options of `collectstatic`. This is used by the `CachedStaticFilesStorage` by default.

Some commonly used options are:

`-noinput`
: Do NOT prompt the user for input of any kind.

`-i` `<pattern>`

`-ignore` `<pattern>`
: Ignore files or directories matching this glob-style pattern. Use multiple times to ignore more.

`-n`

-dry-run
Do everything except modify the filesystem.

-c

-clear
New in version 1.4: *Please see the release notes* Clear the existing files before trying to copy or link the original file.

-l

-link
Create a symbolic link to each file instead of copying.

-no-post-process
New in version 1.4: *Please see the release notes* Don't call the `post_process()` method of the configured `STATICFILES_STORAGE` storage backend.

-no-default-ignore
Don't ignore the common private glob-style patterns `'CVS'`, `'.*'` and `'*~'`.

For a full list of options, refer to the commands own help by running:

```
$ python manage.py collectstatic --help
```

findstatic

django-admin.py findstatic

Searches for one or more relative paths with the enabled finders.

For example:

```
$ python manage.py findstatic css/base.css admin/js/core.js
/home/special.polls.com/core/static/css/base.css
/home/polls.com/core/static/css/base.css
/home/polls.com/src/django/contrib/admin/media/js/core.js
```

By default, all matching locations are found. To only return the first match for each relative path, use the `--first` option:

```
$ python manage.py findstatic css/base.css --first
/home/special.polls.com/core/static/css/base.css
```

This is a debugging aid; it'll show you exactly which static file will be collected for a given path.

runserver

django-admin.py runserver

Overrides the core `runserver` command if the `staticfiles` app is `installed` and adds automatic serving of static files and the following new options.

-nostatic

Use the `--nostatic` option to disable serving of static files with the *staticfiles* app entirely. This option is only available if the *staticfiles* app is in your project's `INSTALLED_APPS` setting.

Example usage:

```
django-admin.py runserver --nostatic
```

-insecure

Use the `--insecure` option to force serving of static files with the *staticfiles* app even if the `DEBUG` setting is `False`. By using this you acknowledge the fact that it's **grossly inefficient** and probably **insecure**. This is only intended for local development, should **never be used in production** and is only available if the *staticfiles* app is in your project's `INSTALLED_APPS` setting.

Example usage:

```
django-admin.py runserver --insecure
```

Storages

StaticFilesStorage

class `storage.`**`StaticFilesStorage`**

A subclass of the `FileSystemStorage` storage backend that uses the `STATIC_ROOT` setting as the base file system location and the `STATIC_URL` setting respectively as the base URL.

`post_process`(*paths*, ***options*)

New in version 1.4: *Please see the release notes* This method is called by the `collectstatic` management command after each run and gets passed the local storages and paths of found files as a dictionary, as well as the command line options.

The `CachedStaticFilesStorage` uses this behind the scenes to replace the paths with their hashed counterparts and update the cache appropriately.

CachedStaticFilesStorage

class `storage.`**`CachedStaticFilesStorage`**

New in version 1.4: *Please see the release notes* A subclass of the `StaticFilesStorage` storage backend which caches the files it saves by appending the MD5 hash of the file's content to the filename. For example, the file `css/styles.css` would also be saved as `css/styles.55e7cbb9ba48.css`.

The purpose of this storage is to keep serving the old files in case some pages still refer to those files, e.g. because they are cached by you or a 3rd party proxy server. Additionally, it's very helpful if you want to apply far future Expires headers to the deployed files to speed up the load time for subsequent page visits.

The storage backend automatically replaces the paths found in the saved files matching other saved files with the path of the cached copy (using the `post_process()` method). The regular expressions used to find those paths (`django.contrib.staticfiles.storage.CachedStaticFilesStorage.cached_patterns`) by default cover the @import rule and url() statement of Cascading Style Sheets. For example, the `'css/styles.css'` file with the content

```
@import url("../admin/css/base.css");
```

would be replaced by calling the `url()` method of the `CachedStaticFilesStorage` storage backend, ultimatively saving a `'css/styles.55e7cbb9ba48.css'` file with the following content:

```
@import url("../admin/css/base.27e20196a850.css");
```

To enable the `CachedStaticFilesStorage` you have to make sure the following requirements are met:

- the `STATICFILES_STORAGE` setting is set to `'django.contrib.staticfiles.storage.CachedStaticFil`
- the `DEBUG` setting is set to `False`
- you use the `staticfiles static` template tag to refer to your static files in your templates
- you've collected all your static files by using the `collectstatic` management command

Since creating the MD5 hash can be a performance burden to your website during runtime, `staticfiles` will automatically try to cache the hashed name for each file path using Django's *caching framework*. If you want to override certain options of the cache backend the storage uses, simply specify a custom entry in the `CACHES` setting named `'staticfiles'`. It falls back to using the `'default'` cache backend.

file_hash(*name*, *content=None*)

New in version 1.5: *Please see the release notes* The method that is used when creating the hashed name of a file. Needs to return a hash for the given file name and content. By default it calculates a MD5 hash from the content's chunks as mentioned above.

Template tags

static

New in version 1.4: *Please see the release notes* Uses the configured `STATICFILES_STORAGE` storage to create the full URL for the given relative path, e.g.:

```
{% load static from staticfiles %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

The previous example is equal to calling the `url` method of an instance of `STATICFILES_STORAGE` with `"images/hi.jpg"`. This is especially useful when using a non-local storage backend to deploy files as documented in *Serving static files from a cloud service or CDN*. New in version 1.5: *Please see the release notes* If you'd like to retrieve a static URL without displaying it, you can use a slightly different call:

```
{% load static from staticfiles %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}" alt="Hi!" />
```

Other Helpers

There are a few other helpers outside of the `staticfiles` app to work with static files:

- The `django.core.context_processors.static()` context processor which adds `STATIC_URL` to every template context rendered with `RequestContext` contexts.
- The builtin template tag `static` which takes a path and urljoins it with the static prefix `STATIC_URL`.
- The builtin template tag `get_static_prefix` which populates a template variable with the static prefix `STATIC_URL` to be used as a variable or directly.
- The similar template tag `get_media_prefix` which works like `get_static_prefix` but uses `MEDIA_URL`.

Static file development view

`django.contrib.staticfiles.views.`**serve**(*request*, *path*)

This view function serves static files in development.

> **Warning:** This view will only work if `DEBUG` is `True`.
> That's because this view is **grossly inefficient** and probably **insecure**. This is only intended for local development, and should **never be used in production**.

This view is automatically enabled by `runserver` (with a `DEBUG` setting set to `True`). To use the view with a different local development server, add the following snippet to the end of your primary URL configuration:

```
from django.conf import settings

if settings.DEBUG:
    urlpatterns += patterns('django.contrib.staticfiles.views',
        url(r'^static/(?P<path>.*)$', 'serve'),
    )
```

Note, the beginning of the pattern (`r'^static/'`) should be your `STATIC_URL` setting.

Since this is a bit finicky, there's also a helper function that'll do this for you:

`django.contrib.staticfiles.urls.`**`staticfiles_urlpatterns`**`()`

This will return the proper URL pattern for serving static files to your already defined pattern list. Use it like this:

```
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

# ... the rest of your URLconf here ...

urlpatterns += staticfiles_urlpatterns()
```

> **Warning:** This helper function will only work if `DEBUG` is `True` and your `STATIC_URL` setting is neither empty nor a full URL such as `http://static.example.com/`.

6.4.18 The syndication feed framework

Django comes with a high-level syndication-feed-generating framework that makes creating RSS and Atom feeds easy.

To create any syndication feed, all you have to do is write a short Python class. You can create as many feeds as you want.

Django also comes with a lower-level feed-generating API. Use this if you want to generate feeds outside of a Web context, or in some other lower-level way.

The high-level framework

Overview

The high-level feed-generating framework is supplied by the `Feed` class. To create a feed, write a `Feed` class and point to an instance of it in your *URLconf*.

Feed classes

A `Feed` class is a Python class that represents a syndication feed. A feed can be simple (e.g., a "site news" feed, or a basic feed displaying the latest entries of a blog) or more complex (e.g., a feed displaying all the blog entries in a particular category, where the category is variable).

Feed classes subclass `django.contrib.syndication.views.Feed`. They can live anywhere in your codebase.

Instances of `Feed` classes are views which can be used in your *URLconf*.

A simple example

This simple example, taken from chicagocrime.org, describes a feed of the latest five news items:

```
from django.contrib.syndication.views import Feed
from chicagocrime.models import NewsItem

class LatestEntriesFeed(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return item.description
```

To connect a URL to this feed, put an instance of the Feed object in your *URLconf*. For example:

```
from django.conf.urls import patterns, url, include
from myproject.feeds import LatestEntriesFeed

urlpatterns = patterns('',
    # ...
    (r'^latest/feed/$', LatestEntriesFeed()),
    # ...
)
```

Note:

- The Feed class subclasses `django.contrib.syndication.views.Feed`.
- `title`, `link` and `description` correspond to the standard RSS `<title>`, `<link>` and `<description>` elements, respectively.
- `items()` is, simply, a method that returns a list of objects that should be included in the feed as `<item>` elements. Although this example returns `NewsItem` objects using Django's *object-relational mapper*, `items()` doesn't have to return model instances. Although you get a few bits of functionality "for free" by using Django models, `items()` can return any type of object you want.
- If you're creating an Atom feed, rather than an RSS feed, set the `subtitle` attribute instead of the `description` attribute. See Publishing Atom and RSS feeds in tandem, later, for an example.

One thing is left to do. In an RSS feed, each <item> has a <title>, <link> and <description>. We need to tell the framework what data to put into those elements.

- For the contents of <title> and <description>, Django tries calling the methods item_title() and item_description() on the Feed class. They are passed a single parameter, item, which is the object itself. These are optional; by default, the unicode representation of the object is used for both.

  If you want to do any special formatting for either the title or description, *Django templates* can be used instead. Their paths can be specified with the title_template and description_template attributes on the Feed class. The templates are rendered for each item and are passed two template context variables:

  - {{ obj }} – The current object (one of whichever objects you returned in items()).
  - {{ site }} – A django.contrib.sites.models.Site object representing the current site. This is useful for {{ site.domain }} or {{ site.name }}. If you do *not* have the Django sites framework installed, this will be set to a django.contrib.sites.models.RequestSite object. See the *RequestSite section of the sites framework documentation* for more.

  See a complex example below that uses a description template.

- To specify the contents of <link>, you have two options. For each item in items(), Django first tries calling the item_link() method on the Feed class. In a similar way to the title and description, it is passed it a single parameter, item. If that method doesn't exist, Django tries executing a get_absolute_url() method on that object. Both get_absolute_url() and item_link() should return the item's URL as a normal Python string. As with get_absolute_url(), the result of item_link() will be included directly in the URL, so you are responsible for doing all necessary URL quoting and conversion to ASCII inside the method itself.

A complex example

The framework also supports more complex feeds, via arguments.

For example, chicagocrime.org offers an RSS feed of recent crimes for every police beat in Chicago. It'd be silly to create a separate Feed class for each police beat; that would violate the *DRY principle* and would couple data to programming logic. Instead, the syndication framework lets you access the arguments passed from your *URLconf* so feeds can output items based on information in the feed's URL.

On chicagocrime.org, the police-beat feeds are accessible via URLs like this:

- /beats/613/rss/ – Returns recent crimes for beat 613.
- /beats/1424/rss/ – Returns recent crimes for beat 1424.

These can be matched with a *URLconf* line such as:

```
(r'^beats/(?P<beat_id>\d+)/rss/$', BeatFeed()),
```

Like a view, the arguments in the URL are passed to the get_object() method along with the request object.

Here's the code for these beat-specific feeds:

```
from django.contrib.syndication.views import FeedDoesNotExist
from django.shortcuts import get_object_or_404

class BeatFeed(Feed):
    description_template = 'feeds/beat_description.html'

    def get_object(self, request, beat_id):
        return get_object_or_404(Beat, pk=beat_id)

    def title(self, obj):
```

```
        return "Chicagocrime.org: Crimes for beat %s" % obj.beat

    def link(self, obj):
        return obj.get_absolute_url()

    def description(self, obj):
        return "Crimes recently reported in police beat %s" % obj.beat

    def items(self, obj):
        return Crime.objects.filter(beat=obj).order_by('-crime_date')[:30]
```

To generate the feed's <title>, <link> and <description>, Django uses the `title()`, `link()` and `description()` methods. In the previous example, they were simple string class attributes, but this example illustrates that they can be either strings *or* methods. For each of `title`, `link` and `description`, Django follows this algorithm:

- First, it tries to call a method, passing the `obj` argument, where `obj` is the object returned by `get_object()`.
- Failing that, it tries to call a method with no arguments.
- Failing that, it uses the class attribute.

Also note that `items()` also follows the same algorithm – first, it tries `items(obj)`, then `items()`, then finally an `items` class attribute (which should be a list).

We are using a template for the item descriptions. It can be very simple:

```
{{ obj.description }}
```

However, you are free to add formatting as desired.

The `ExampleFeed` class below gives full documentation on methods and attributes of `Feed` classes.

Specifying the type of feed

By default, feeds produced in this framework use RSS 2.0.

To change that, add a `feed_type` attribute to your `Feed` class, like so:

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

Note that you set `feed_type` to a class object, not an instance.

Currently available feed types are:

- `django.utils.feedgenerator.Rss201rev2Feed` (RSS 2.01. Default.)
- `django.utils.feedgenerator.RssUserland091Feed` (RSS 0.91.)
- `django.utils.feedgenerator.Atom1Feed` (Atom 1.0.)

Enclosures

To specify enclosures, such as those used in creating podcast feeds, use the `item_enclosure_url`, `item_enclosure_length` and `item_enclosure_mime_type` hooks. See the `ExampleFeed` class below for usage examples.

Language

Feeds created by the syndication framework automatically include the appropriate `<language>` tag (RSS 2.0) or `xml:lang` attribute (Atom). This comes directly from your `LANGUAGE_CODE` setting.

URLs

The `link` method/attribute can return either an absolute path (e.g. `"/blog/"`) or a URL with the fully-qualified domain and protocol (e.g. `"http://www.example.com/blog/"`). If `link` doesn't return the domain, the syndication framework will insert the domain of the current site, according to your `SITE_ID setting`.

Atom feeds require a `<link rel="self">` that defines the feed's current location. The syndication framework populates this automatically, using the domain of the current site according to the `SITE_ID` setting.

Publishing Atom and RSS feeds in tandem

Some developers like to make available both Atom *and* RSS versions of their feeds. That's easy to do with Django: Just create a subclass of your `Feed` class and set the `feed_type` to something different. Then update your URLconf to add the extra versions.

Here's a full example:

```
from django.contrib.syndication.views import Feed
from chicagocrime.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Chicagocrime.org site news"
    link = "/sitenews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
    subtitle = RssSiteNewsFeed.description
```

Note: In this example, the RSS feed uses a `description` while the Atom feed uses a `subtitle`. That's because Atom feeds don't provide for a feed-level "description," but they *do* provide for a "subtitle."

If you provide a `description` in your `Feed` class, Django will *not* automatically put that into the `subtitle` element, because a subtitle and description are not necessarily the same thing. Instead, you should define a `subtitle` attribute.

In the above example, we simply set the Atom feed's `subtitle` to the RSS feed's `description`, because it's quite short already.

And the accompanying URLconf:

```
from django.conf.urls import patterns, url, include
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

urlpatterns = patterns('',
    # ...
```

```
    (r'^sitenews/rss/$', RssSiteNewsFeed()),
    (r'^sitenews/atom/$', AtomSiteNewsFeed()),
    # ...
)
```

Feed class reference

class `django.contrib.syndication.views.`**Feed**

This example illustrates all possible attributes and methods for a `Feed` class:

```
from django.contrib.syndication.views import Feed
from django.utils import feedgenerator

class ExampleFeed(Feed):

    # FEED TYPE -- Optional. This should be a class that subclasses
    # django.utils.feedgenerator.SyndicationFeed. This designates
    # which type of feed this should be: RSS 2.0, Atom 1.0, etc. If
    # you don't specify feed_type, your feed will be RSS 2.0. This
    # should be a class, not an instance of the class.

    feed_type = feedgenerator.Rss201rev2Feed

    # TEMPLATE NAMES -- Optional. These should be strings
    # representing names of Django templates that the system should
    # use in rendering the title and description of your feed items.
    # Both are optional. If a template is not specified, the
    # item_title() or item_description() methods are used instead.

    title_template = None
    description_template = None

    # TITLE -- One of the following three is required. The framework
    # looks for them in this order.

    def title(self, obj):
        """
        Takes the object returned by get_object() and returns the
        feed's title as a normal Python string.
        """

    def title(self):
        """
        Returns the feed's title as a normal Python string.
        """

    title = 'foo' # Hard-coded title.

    # LINK -- One of the following three is required. The framework
    # looks for them in this order.

    def link(self, obj):
        """
        # Takes the object returned by get_object() and returns the URL
        # of the HTML version of the feed as a normal Python string.
        """
```

```
    def link(self):
        """
        Returns the URL of the HTML version of the feed as a normal Python
        string.
        """

    link = '/blog/' # Hard-coded URL.

    # FEED_URL -- One of the following three is optional. The framework
    # looks for them in this order.

    def feed_url(self, obj):
        """
        # Takes the object returned by get_object() and returns the feed's
        # own URL as a normal Python string.
        """

    def feed_url(self):
        """
        Returns the feed's own URL as a normal Python string.
        """

    feed_url = '/blog/rss/' # Hard-coded URL.

    # GUID -- One of the following three is optional. The framework looks
    # for them in this order. This property is only used for Atom feeds
    # (where it is the feed-level ID element). If not provided, the feed
    # link is used as the ID.

    def feed_guid(self, obj):
        """
        Takes the object returned by get_object() and returns the globally
        unique ID for the feed as a normal Python string.
        """

    def feed_guid(self):
        """
        Returns the feed's globally unique ID as a normal Python string.
        """

    feed_guid = '/foo/bar/1234' # Hard-coded guid.

    # DESCRIPTION -- One of the following three is required. The framework
    # looks for them in this order.

    def description(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        description as a normal Python string.
        """

    def description(self):
        """
        Returns the feed's description as a normal Python string.
        """

    description = 'Foo bar baz.' # Hard-coded description.
```

```
    # AUTHOR NAME --One of the following three is optional. The framework
    # looks for them in this order.

    def author_name(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        author's name as a normal Python string.
        """

    def author_name(self):
        """
        Returns the feed's author's name as a normal Python string.
        """

    author_name = 'Sally Smith' # Hard-coded author name.

    # AUTHOR E-MAIL --One of the following three is optional. The framework
    # looks for them in this order.

    def author_email(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        author's email as a normal Python string.
        """

    def author_email(self):
        """
        Returns the feed's author's email as a normal Python string.
        """

    author_email = 'test@example.com' # Hard-coded author email.

    # AUTHOR LINK --One of the following three is optional. The framework
    # looks for them in this order. In each case, the URL should include
    # the "http://" and domain name.

    def author_link(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        author's URL as a normal Python string.
        """

    def author_link(self):
        """
        Returns the feed's author's URL as a normal Python string.
        """

    author_link = 'http://www.example.com/' # Hard-coded author URL.

    # CATEGORIES -- One of the following three is optional. The framework
    # looks for them in this order. In each case, the method/attribute
    # should return an iterable object that returns strings.

    def categories(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        categories as iterable over strings.
        """
```

```
    def categories(self):
        """
        Returns the feed's categories as iterable over strings.
        """

    categories = ("python", "django") # Hard-coded list of categories.

    # COPYRIGHT NOTICE -- One of the following three is optional. The
    # framework looks for them in this order.

    def feed_copyright(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        copyright notice as a normal Python string.
        """

    def feed_copyright(self):
        """
        Returns the feed's copyright notice as a normal Python string.
        """

    feed_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright notice.

    # TTL -- One of the following three is optional. The framework looks
    # for them in this order. Ignored for Atom feeds.

    def ttl(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        TTL (Time To Live) as a normal Python string.
        """

    def ttl(self):
        """
        Returns the feed's TTL as a normal Python string.
        """

    ttl = 600 # Hard-coded Time To Live.

    # ITEMS -- One of the following three is required. The framework looks
    # for them in this order.

    def items(self, obj):
        """
        Takes the object returned by get_object() and returns a list of
        items to publish in this feed.
        """

    def items(self):
        """
        Returns a list of items to publish in this feed.
        """

    items = ('Item 1', 'Item 2') # Hard-coded items.

    # GET_OBJECT -- This is required for feeds that publish different data
    # for different URL parameters. (See "A complex example" above.)
```

```
    def get_object(self, request, *args, **kwargs):
        """
        Takes the current request and the arguments from the URL, and
        returns an object represented by this feed. Raises
        django.core.exceptions.ObjectDoesNotExist on error.
        """

    # ITEM TITLE AND DESCRIPTION -- If title_template or
    # description_template are not defined, these are used instead. Both are
    # optional, by default they will use the unicode representation of the
    # item.

    def item_title(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        title as a normal Python string.
        """

    def item_title(self):
        """
        Returns the title for every item in the feed.
        """

    item_title = 'Breaking News: Nothing Happening' # Hard-coded title.

    def item_description(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        description as a normal Python string.
        """

    def item_description(self):
        """
        Returns the description for every item in the feed.
        """

    item_description = 'A description of the item.' # Hard-coded description.

    # ITEM LINK -- One of these three is required. The framework looks for
    # them in this order.

    # First, the framework tries the two methods below, in
    # order. Failing that, it falls back to the get_absolute_url()
    # method on each item returned by items().

    def item_link(self, item):
        """
        Takes an item, as returned by items(), and returns the item's URL.
        """

    def item_link(self):
        """
        Returns the URL for every item in the feed.
        """

    # ITEM_GUID -- The following method is optional. If not provided, the
    # item's link is used by default.
```

```
    def item_guid(self, obj):
        """
        Takes an item, as return by items(), and returns the item's ID.
        """

    # ITEM AUTHOR NAME -- One of the following three is optional. The
    # framework looks for them in this order.

    def item_author_name(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        author's name as a normal Python string.
        """

    def item_author_name(self):
        """
        Returns the author name for every item in the feed.
        """

    item_author_name = 'Sally Smith' # Hard-coded author name.

    # ITEM AUTHOR E-MAIL --One of the following three is optional. The
    # framework looks for them in this order.
    #
    # If you specify this, you must specify item_author_name.

    def item_author_email(self, obj):
        """
        Takes an item, as returned by items(), and returns the item's
        author's email as a normal Python string.
        """

    def item_author_email(self):
        """
        Returns the author email for every item in the feed.
        """

    item_author_email = 'test@example.com' # Hard-coded author email.

    # ITEM AUTHOR LINK -- One of the following three is optional. The
    # framework looks for them in this order. In each case, the URL should
    # include the "http://" and domain name.
    #
    # If you specify this, you must specify item_author_name.

    def item_author_link(self, obj):
        """
        Takes an item, as returned by items(), and returns the item's
        author's URL as a normal Python string.
        """

    def item_author_link(self):
        """
        Returns the author URL for every item in the feed.
        """

    item_author_link = 'http://www.example.com/' # Hard-coded author URL.
```

```
    # ITEM ENCLOSURE URL -- One of these three is required if you're
    # publishing enclosures. The framework looks for them in this order.

    def item_enclosure_url(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        enclosure URL.
        """

    def item_enclosure_url(self):
        """
        Returns the enclosure URL for every item in the feed.
        """

    item_enclosure_url = "/foo/bar.mp3" # Hard-coded enclosure link.

    # ITEM ENCLOSURE LENGTH -- One of these three is required if you're
    # publishing enclosures. The framework looks for them in this order.
    # In each case, the returned value should be either an integer, or a
    # string representation of the integer, in bytes.

    def item_enclosure_length(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        enclosure length.
        """

    def item_enclosure_length(self):
        """
        Returns the enclosure length for every item in the feed.
        """

    item_enclosure_length = 32000 # Hard-coded enclosure length.

    # ITEM ENCLOSURE MIME TYPE -- One of these three is required if you're
    # publishing enclosures. The framework looks for them in this order.

    def item_enclosure_mime_type(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        enclosure MIME type.
        """

    def item_enclosure_mime_type(self):
        """
        Returns the enclosure MIME type for every item in the feed.
        """

    item_enclosure_mime_type = "audio/mpeg" # Hard-coded enclosure MIME type.

    # ITEM PUBDATE -- It's optional to use one of these three. This is a
    # hook that specifies how to get the pubdate for a given item.
    # In each case, the method/attribute should return a Python
    # datetime.datetime object.

    def item_pubdate(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
```

```
        pubdate.
        """

    def item_pubdate(self):
        """
        Returns the pubdate for every item in the feed.
        """

    item_pubdate = datetime.datetime(2005, 5, 3) # Hard-coded pubdate.

    # ITEM CATEGORIES -- It's optional to use one of these three. This is
    # a hook that specifies how to get the list of categories for a given
    # item. In each case, the method/attribute should return an iterable
    # object that returns strings.

    def item_categories(self, item):
        """
        Takes an item, as returned by items(), and returns the item's
        categories.
        """

    def item_categories(self):
        """
        Returns the categories for every item in the feed.
        """

    item_categories = ("python", "django") # Hard-coded categories.

    # ITEM COPYRIGHT NOTICE (only applicable to Atom feeds) -- One of the
    # following three is optional. The framework looks for them in this
    # order.

    def item_copyright(self, obj):
        """
        Takes an item, as returned by items(), and returns the item's
        copyright notice as a normal Python string.
        """

    def item_copyright(self):
        """
        Returns the copyright notice for every item in the feed.
        """

    item_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright notice.
```

The low-level framework

Behind the scenes, the high-level RSS framework uses a lower-level framework for generating feeds' XML. This framework lives in a single module: django/utils/feedgenerator.py.

You use this framework on your own, for lower-level feed generation. You can also create custom feed generator subclasses for use with the `feed_type` `Feed` option.

`SyndicationFeed` classes

The `feedgenerator` module contains a base class:

- `django.utils.feedgenerator.SyndicationFeed`

and several subclasses:

- `django.utils.feedgenerator.RssUserland091Feed`
- `django.utils.feedgenerator.Rss201rev2Feed`
- `django.utils.feedgenerator.Atom1Feed`

Each of these three classes knows how to render a certain type of feed as XML. They share this interface:

`SyndicationFeed.__init__()` Initialize the feed with the given dictionary of metadata, which applies to the entire feed. Required keyword arguments are:

- `title`
- `link`
- `description`

There's also a bunch of other optional keywords:

- `language`
- `author_email`
- `author_name`
- `author_link`
- `subtitle`
- `categories`
- `feed_url`
- `feed_copyright`
- `feed_guid`
- `ttl`

Any extra keyword arguments you pass to `__init__` will be stored in `self.feed` for use with custom feed generators.

All parameters should be Unicode objects, except `categories`, which should be a sequence of Unicode objects.

`SyndicationFeed.add_item()` Add an item to the feed with the given parameters.

Required keyword arguments are:

- `title`
- `link`
- `description`

Optional keyword arguments are:

- `author_email`
- `author_name`
- `author_link`
- `pubdate`
- `comments`

- unique_id
- enclosure
- categories
- item_copyright
- ttl

Extra keyword arguments will be stored for custom feed generators.

All parameters, if given, should be Unicode objects, except:

- pubdate should be a Python datetime object.
- enclosure should be an instance of django.utils.feedgenerator.Enclosure.
- categories should be a sequence of Unicode objects.

SyndicationFeed.write() Outputs the feed in the given encoding to outfile, which is a file-like object.

SyndicationFeed.writeString() Returns the feed as a string in the given encoding.

For example, to create an Atom 1.0 feed and print it to standard output:

```
>>> from django.utils import feedgenerator
>>> from datetime import datetime
>>> f = feedgenerator.Atom1Feed(
...     title=u"My Weblog",
...     link=u"http://www.example.com/",
...     description=u"In which I write about what I ate today.",
...     language=u"en",
...     author_name=u"Myself",
...     feed_url=u"http://example.com/atom.xml")
>>> f.add_item(title=u"Hot dog today",
...     link=u"http://www.example.com/entries/1/",
...     pubdate=datetime.now(),
...     description=u"<p>Today I had a Vienna Beef hot dog. It was pink, plump and perfect.</p>")
>>> print(f.writeString('UTF-8'))
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
...
</feed>
```

Custom feed generators

If you need to produce a custom feed format, you've got a couple of options.

If the feed format is totally custom, you'll want to subclass SyndicationFeed and completely replace the write() and writeString() methods.

However, if the feed format is a spin-off of RSS or Atom (i.e. GeoRSS, Apple's iTunes podcast format, etc.), you've got a better choice. These types of feeds typically add extra elements and/or attributes to the underlying format, and there are a set of methods that SyndicationFeed calls to get these extra attributes. Thus, you can subclass the appropriate feed generator class (Atom1Feed or Rss201rev2Feed) and extend these callbacks. They are:

SyndicationFeed.root_attributes(self,) Return a dict of attributes to add to the root feed element (feed/channel).

SyndicationFeed.add_root_elements(self, handler) Callback to add elements inside the root feed element (feed/channel). handler is an XMLGenerator from Python's built-in SAX library; you'll call methods on it to add to the XML document in process.

SyndicationFeed.item_attributes(self, item) Return a `dict` of attributes to add to each item (`item`/`entry`) element. The argument, `item`, is a dictionary of all the data passed to `SyndicationFeed.add_item()`.

SyndicationFeed.add_item_elements(self, handler, item) Callback to add elements to each item (`item`/`entry`) element. `handler` and `item` are as above.

> **Warning:** If you override any of these methods, be sure to call the superclass methods since they add the required elements for each feed format.

For example, you might start implementing an iTunes RSS feed generator like so:

```
class iTunesFeed(Rss201rev2Feed):
    def root_attributes(self):
        attrs = super(iTunesFeed, self).root_attributes()
        attrs['xmlns:itunes'] = 'http://www.itunes.com/dtds/podcast-1.0.dtd'
        return attrs

    def add_root_elements(self, handler):
        super(iTunesFeed, self).add_root_elements(handler)
        handler.addQuickElement('itunes:explicit', 'clean')
```

Obviously there's a lot more work to be done for a complete custom feed class, but the above example should demonstrate the basic idea.

6.4.19 django.contrib.webdesign

The `django.contrib.webdesign` package, part of the *"django.contrib" add-ons*, provides various Django helpers that are particularly useful to Web *designers* (as opposed to developers).

At present, the package contains only a single template tag. If you have ideas for Web-designer-friendly functionality in Django, please *suggest them*.

Template tags

To use these template tags, add `'django.contrib.webdesign'` to your `INSTALLED_APPS` setting. Once you've done that, use `{% load webdesign %}` in a template to give your template access to the tags.

lorem

Displays random "lorem ipsum" Latin text. This is useful for providing sample data in templates.

Usage:

```
{% lorem [count] [method] [random] %}
```

The `{% lorem %}` tag can be used with zero, one, two or three arguments. The arguments are:

Argument	Description
`count`	A number (or variable) containing the number of paragraphs or words to generate (default is 1).
`method`	Either `w` for words, `p` for HTML paragraphs or `b` for plain-text paragraph blocks (default is `b`).
`random`	The word `random`, which if given, does not use the common paragraph ("Lorem ipsum dolor sit amet...") when generating text.

Examples:

- `{% lorem %}` will output the common "lorem ipsum" paragraph.
- `{% lorem 3 p %}` will output the common "lorem ipsum" paragraph and two random paragraphs each wrapped in HTML <p> tags.
- `{% lorem 2 w random %}` will output two random Latin words.

6.4.20 admin

The automatic Django administrative interface. For more information, see *Tutorial 2* and the *admin documentation*.

Requires the auth and contenttypes contrib packages to be installed.

6.4.21 auth

Django's authentication framework.

See *User authentication in Django*.

6.4.22 comments

A simple yet flexible comments system. See *Django's comments framework*.

6.4.23 contenttypes

A light framework for hooking into "types" of content, where each installed Django model is a separate content type.

See the *contenttypes documentation*.

6.4.24 csrf

A middleware for preventing Cross Site Request Forgeries

See the *csrf documentation*.

6.4.25 flatpages

A framework for managing simple "flat" HTML content in a database.

See the *flatpages documentation*.

Requires the sites contrib package to be installed as well.

6.4.26 formtools

A set of high-level abstractions for Django forms (django.forms).

django.contrib.formtools.preview

An abstraction of the following workflow:

"Display an HTML form, force a preview, then do something with the submission."

See the *form preview documentation*.

django.contrib.formtools.wizard

Splits forms across multiple Web pages.

See the *form wizard documentation*.

6.4.27 gis

A world-class geospatial framework built on top of Django, that enables storage, manipulation and display of spatial data.

See the *GeoDjango* documentation for more.

6.4.28 humanize

A set of Django template filters useful for adding a "human touch" to data.

See the *humanize documentation*.

6.4.29 localflavor

A collection of various Django snippets that are useful only for a particular country or culture. For example, `django.contrib.localflavor.us.forms` contains a `USZipCodeField` that you can use to validate U.S. zip codes.

See the *localflavor documentation*.

6.4.30 markup

A collection of template filters that implement common markup languages

See the *markup documentation*.

6.4.31 messages

A framework for storing and retrieving temporary cookie- or session-based messages

See the *messages documentation*.

6.4.32 redirects

A framework for managing redirects.

See the *redirects documentation*.

6.4.33 sessions

A framework for storing data in anonymous sessions.

See the *sessions documentation*.

6.4.34 sites

A light framework that lets you operate multiple Web sites off of the same database and Django installation. It gives you hooks for associating objects to one or more sites.

See the *sites documentation*.

6.4.35 sitemaps

A framework for generating Google sitemap XML files.

See the *sitemaps documentation*.

6.4.36 syndication

A framework for generating syndication feeds, in RSS and Atom, quite easily.

See the *syndication documentation*.

6.4.37 webdesign

Helpers and utilities targeted primarily at Web *designers* rather than Web *developers*.

See the *Web design helpers documentation*.

6.4.38 Other add-ons

If you have an idea for functionality to include in `contrib`, let us know! Code it up, and post it to the django-users mailing list.

6.5 Databases

Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and we've had to make design decisions on which features to support and which assumptions we can make safely.

This file describes some of the features that might be relevant to Django usage. Of course, it is not intended as a replacement for server-specific documentation or reference manuals.

6.5.1 PostgreSQL notes

Changed in version 1.4: *Please see the release notes* Django supports PostgreSQL 8.2 and higher.

PostgreSQL 8.2 to 8.2.4

The implementation of the population statistics aggregates `STDDEV_POP` and `VAR_POP` that shipped with PostgreSQL 8.2 to 8.2.4 are known to be faulty. Users of these releases of PostgreSQL are advised to upgrade to Release 8.2.5 or later. Django will raise a `NotImplementedError` if you attempt to use the `StdDev(sample=False)` or `Variance(sample=False)` aggregate with a database backend that falls within the affected release range.

Optimizing PostgreSQL's configuration

Django needs the following parameters for its database connections:

- `client_encoding`: `'UTF8'`,
- `default_transaction_isolation`: `'read committed'`,
- `timezone`: `'UTC'` when `USE_TZ` is `True`, value of `TIME_ZONE` otherwise.

If these parameters already have the correct values, Django won't set them for every new connection, which improves performance slightly. You can configure them directly in `postgresql.conf` or more conveniently per database user with ALTER ROLE.

Django will work just fine without this optimization, but each new connection will do some additional queries to set these parameters.

Transaction handling

By default, Django starts a transaction when a database connection is first used and commits the result at the end of the request/response handling. The PostgreSQL backends normally operate the same as any other Django backend in this respect.

Autocommit mode

If your application is particularly read-heavy and doesn't make many database writes, the overhead of a constantly open transaction can sometimes be noticeable. For those situations, you can configure Django to use *"autocommit"* behavior for the connection, meaning that each database operation will normally be in its own transaction, rather than having the transaction extend over multiple operations. In this case, you can still manually start a transaction if you're doing something that requires consistency across multiple database operations. The autocommit behavior is enabled by setting the `autocommit` key in the `OPTIONS` part of your database configuration in `DATABASES`:

```
'OPTIONS': {
    'autocommit': True,
}
```

In this configuration, Django still ensures that *delete()* and *update()* queries run inside a single transaction, so that either all the affected objects are changed or none of them are.

This is database-level autocommit

This functionality is not the same as the *autocommit* decorator. That decorator is a Django-level implementation that commits automatically after data changing operations. The feature enabled using the `OPTIONS` option provides autocommit behavior at the database adapter level. It commits after *every* operation.

If you are using this feature and performing an operation akin to delete or updating that requires multiple operations, you are strongly recommended to wrap you operations in manual transaction handling to ensure data consistency. You

should also audit your existing code for any instances of this behavior before enabling this feature. It's faster, but it provides less automatic protection for multi-call operations.

Indexes for `varchar` and `text` columns

When specifying `db_index=True` on your model fields, Django typically outputs a single `CREATE INDEX` statement. However, if the database type for the field is either `varchar` or `text` (e.g., used by `CharField`, `FileField`, and `TextField`), then Django will create an additional index that uses an appropriate PostgreSQL operator class for the column. The extra index is necessary to correctly perform lookups that use the `LIKE` operator in their SQL, as is done with the `contains` and `startswith` lookup types.

6.5.2 MySQL notes

Version support

Django supports MySQL 5.0.3 and higher.

MySQL 5.0 adds the `information_schema` database, which contains detailed data on all database schema. Django's `inspectdb` feature uses it. Changed in version 1.5: The minimum version requirement of MySQL 5.0.3 was set in Django 1.5. Django expects the database to support Unicode (UTF-8 encoding) and delegates to it the task of enforcing transactions and referential integrity. It is important to be aware of the fact that the two latter ones aren't actually enforced by MySQL when using the MyISAM storage engine, see the next section.

Storage engines

MySQL has several storage engines (previously called table types). You can change the default storage engine in the server configuration.

Until MySQL 5.5.4, the default engine was MyISAM [33]. The main drawbacks of MyISAM are that it doesn't support transactions or enforce foreign-key constraints. On the plus side, it's currently the only engine that supports full-text indexing and searching.

Since MySQL 5.5.5, the default storage engine is InnoDB. This engine is fully transactional and supports foreign key references. It's probably the best choice at this point. Changed in version 1.4: *Please see the release notes* In previous versions of Django, fixtures with forward references (i.e. relations to rows that have not yet been inserted into the database) would fail to load when using the InnoDB storage engine. This was due to the fact that InnoDB deviates from the SQL standard by checking foreign key constraints immediately instead of deferring the check until the transaction is committed. This problem has been resolved in Django 1.4. Fixture data is now loaded with foreign key checks turned off; foreign key checks are then re-enabled when the data has finished loading, at which point the entire table is checked for invalid foreign key references and an *IntegrityError* is raised if any are found.

MySQLdb

MySQLdb is the Python interface to MySQL. Version 1.2.1p2 or later is required for full MySQL support in Django.

Note: If you see `ImportError: cannot import name ImmutableSet` when trying to use Django, your MySQLdb installation may contain an outdated `sets.py` file that conflicts with the built-in module of the same name from Python 2.4 and later. To fix this, verify that you have installed MySQLdb version 1.2.1p2 or newer, then delete the `sets.py` file in the MySQLdb directory that was left by an earlier version.

---

[33] Unless this was changed by the packager of your MySQL package. We've had reports that the Windows Community Server installer sets up InnoDB as the default storage engine, for example.

Creating your database

You can create your database using the command-line tools and this SQL:

```
CREATE DATABASE <dbname> CHARACTER SET utf8;
```

This ensures all tables and columns will use UTF-8 by default.

Collation settings

The collation setting for a column controls the order in which data is sorted as well as what strings compare as equal. It can be set on a database-wide level and also per-table and per-column. This is documented thoroughly in the MySQL documentation. In all cases, you set the collation by directly manipulating the database tables; Django doesn't provide a way to set this on the model definition.

By default, with a UTF-8 database, MySQL will use the `utf8_general_ci_swedish` collation. This results in all string equality comparisons being done in a *case-insensitive* manner. That is, `"Fred"` and `"freD"` are considered equal at the database level. If you have a unique constraint on a field, it would be illegal to try to insert both `"aa"` and `"AA"` into the same column, since they compare as equal (and, hence, non-unique) with the default collation.

In many cases, this default will not be a problem. However, if you really want case-sensitive comparisons on a particular column or table, you would change the column or table to use the `utf8_bin` collation. The main thing to be aware of in this case is that if you are using MySQLdb 1.2.2, the database backend in Django will then return bytestrings (instead of unicode strings) for any character fields it receive from the database. This is a strong variation from Django's normal practice of *always* returning unicode strings. It is up to you, the developer, to handle the fact that you will receive bytestrings if you configure your table(s) to use `utf8_bin` collation. Django itself should mostly work smoothly with such columns (except for the `contrib.sessions Session` and `contrib.admin LogEntry` tables described below), but your code must be prepared to call `django.utils.encoding.smart_unicode()` at times if it really wants to work with consistent data – Django will not do this for you (the database backend layer and the model population layer are separated internally so the database layer doesn't know it needs to make this conversion in this one particular case).

If you're using MySQLdb 1.2.1p2, Django's standard `CharField` class will return unicode strings even with `utf8_bin` collation. However, `TextField` fields will be returned as an `array.array` instance (from Python's standard `array` module). There isn't a lot Django can do about that, since, again, the information needed to make the necessary conversions isn't available when the data is read in from the database. This problem was fixed in MySQLdb 1.2.2, so if you want to use `TextField` with `utf8_bin` collation, upgrading to version 1.2.2 and then dealing with the bytestrings (which shouldn't be too difficult) as described above is the recommended solution.

Should you decide to use `utf8_bin` collation for some of your tables with MySQLdb 1.2.1p2 or 1.2.2, you should still use `utf8_collation_ci_swedish` (the default) collation for the `django.contrib.sessions.models.Session` table (usually called `django_session`) and the `django.contrib.admin.models.LogEntry` table (usually called `django_admin_log`). Those are the two standard tables that use `TextField` internally.

Connecting to the database

Refer to the *settings documentation*.

Connection settings are used in this order:

1. `OPTIONS`.
2. `NAME`, `USER`, `PASSWORD`, `HOST`, `PORT`
3. MySQL option files.

In other words, if you set the name of the database in `OPTIONS`, this will take precedence over `NAME`, which would override anything in a MySQL option file.

Here's a sample configuration which uses a MySQL option file:

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'OPTIONS': {
            'read_default_file': '/path/to/my.cnf',
        },
    }
}


# my.cnf
[client]
database = NAME
user = USER
password = PASSWORD
default-character-set = utf8
```

Several other MySQLdb connection options may be useful, such as `ssl`, `use_unicode`, `init_command`, and `sql_mode`. Consult the MySQLdb documentation for more details.

Creating your tables

When Django generates the schema, it doesn't specify a storage engine, so tables will be created with whatever default storage engine your database server is configured for. The easiest solution is to set your database server's default storage engine to the desired engine.

If you're using a hosting service and can't change your server's default storage engine, you have a couple of options.

- After the tables are created, execute an `ALTER TABLE` statement to convert a table to a new storage engine (such as InnoDB):

  ```
  ALTER TABLE <tablename> ENGINE=INNODB;
  ```

  This can be tedious if you have a lot of tables.

- Another option is to use the `init_command` option for MySQLdb prior to creating your tables:

  ```
  'OPTIONS': {
     'init_command': 'SET storage_engine=INNODB',
  }
  ```

  This sets the default storage engine upon connecting to the database. After your tables have been created, you should remove this option as it adds a query that is only needed during table creation to each database connection.

- Another method for changing the storage engine is described in AlterModelOnSyncDB.

Table names

There are known issues in even the latest versions of MySQL that can cause the case of a table name to be altered when certain SQL statements are executed under certain conditions. It is recommended that you use lowercase table names, if possible, to avoid any problems that might arise from this behavior. Django uses lowercase table names

when it auto-generates table names from models, so this is mainly a consideration if you are overriding the table name via the `db_table` parameter.

Savepoints

Both the Django ORM and MySQL (when using the InnoDB *storage engine*) support database *savepoints*, but this feature wasn't available in Django until version 1.4 when such supports was added.

If you use the MyISAM storage engine please be aware of the fact that you will receive database-generated errors if you try to use the *savepoint-related methods of the transactions API*. The reason for this is that detecting the storage engine of a MySQL database/table is an expensive operation so it was decided it isn't worth to dynamically convert these methods in no-op's based in the results of such detection.

Notes on specific fields

Character fields

Any fields that are stored with `VARCHAR` column types have their `max_length` restricted to 255 characters if you are using `unique=True` for the field. This affects `CharField`, `SlugField` and `CommaSeparatedIntegerField`.

DateTime fields

MySQL does not have a timezone-aware column type. If an attempt is made to store a timezone-aware `time` or `datetime` to a `TimeField` or `DateTimeField` respectively, a `ValueError` is raised rather than truncating data.

MySQL does not store fractions of seconds. Fractions of seconds are truncated to zero when the time is stored.

Row locking with `QuerySet.select_for_update()`

MySQL does not support the `NOWAIT` option to the `SELECT ... FOR UPDATE` statement. If `select_for_update()` is used with `nowait=True` then a `DatabaseError` will be raised.

6.5.3 SQLite notes

SQLite provides an excellent development alternative for applications that are predominantly read-only or require a smaller installation footprint. As with all database servers, though, there are some differences that are specific to SQLite that you should be aware of.

Substring matching and case sensitivity

For all SQLite versions, there is some slightly counter-intuitive behavior when attempting to match some types of strings. These are triggered when using the `iexact` or `contains` filters in Querysets. The behavior splits into two cases:

1. For substring matching, all matches are done case-insensitively. That is a filter such as `filter(name__contains="aa")` will match a name of `"Aabb"`.

2. For strings containing characters outside the ASCII range, all exact string matches are performed case-sensitively, even when the case-insensitive options are passed into the query. So the `iexact` filter will behave exactly the same as the `exact` filter in these cases.

Some possible workarounds for this are documented at sqlite.org, but they aren't utilised by the default SQLite backend in Django, as incorporating them would be fairly difficult to do robustly. Thus, Django exposes the default SQLite behavior and you should be aware of this when doing case-insensitive or substring filtering.

SQLite 3.3.6 or newer strongly recommended

Versions of SQLite 3.3.5 and older contains the following bugs:

- A bug when handling `ORDER BY` parameters. This can cause problems when you use the `select` parameter for the `extra()` QuerySet method. The bug can be identified by the error message `OperationalError: ORDER BY terms must not be non-integer constants`.
- A bug when handling aggregation together with DateFields and DecimalFields.

SQLite 3.3.6 was released in April 2006, so most current binary distributions for different platforms include newer version of SQLite usable from Python through either the `pysqlite2` or the `sqlite3` modules.

Version 3.5.9

The Ubuntu "Intrepid Ibex" (8.10) SQLite 3.5.9-3 package contains a bug that causes problems with the evaluation of query expressions. If you are using Ubuntu "Intrepid Ibex", you will need to update the package to version 3.5.9-3ubuntu1 or newer (recommended) or find an alternate source for SQLite packages, or install SQLite from source.

At one time, Debian Lenny shipped with the same malfunctioning SQLite 3.5.9-3 package. However the Debian project has subsequently issued updated versions of the SQLite package that correct these bugs. If you find you are getting unexpected results under Debian, ensure you have updated your SQLite package to 3.5.9-5 or later.

The problem does not appear to exist with other versions of SQLite packaged with other operating systems.

Version 3.6.2

SQLite version 3.6.2 (released August 30, 2008) introduced a bug into `SELECT DISTINCT` handling that is triggered by, amongst other things, Django's `DateQuerySet` (returned by the `dates()` method on a queryset).

You should avoid using this version of SQLite with Django. Either upgrade to 3.6.3 (released September 22, 2008) or later, or downgrade to an earlier version of SQLite.

Using newer versions of the SQLite DB-API 2.0 driver

For versions of Python 2.5 or newer that include `sqlite3` in the standard library Django will now use a `pysqlite2` interface in preference to `sqlite3` if it finds one is available.

This provides the ability to upgrade both the DB-API 2.0 interface or SQLite 3 itself to versions newer than the ones included with your particular Python binary distribution, if needed.

"Database is locked" errors

SQLite is meant to be a lightweight database, and thus can't support a high level of concurrency. `OperationalError: database is locked` errors indicate that your application is experiencing more concurrency than `sqlite` can handle in default configuration. This error means that one thread or process has an exclusive lock on the database connection and another thread timed out waiting for the lock the be released.

Python's SQLite wrapper has a default timeout value that determines how long the second thread is allowed to wait on the lock before it times out and raises the `OperationalError: database is locked` error.

If you're getting this error, you can solve it by:

- Switching to another database backend. At a certain point SQLite becomes too "lite" for real-world applications, and these sorts of concurrency errors indicate you've reached that point.
- Rewriting your code to reduce concurrency and ensure that database transactions are short-lived.
- Increase the default timeout value by setting the `timeout` database option option:

```
'OPTIONS': {
    # ...
    'timeout': 20,
    # ...
}
```

  This will simply make SQLite wait a bit longer before throwing "database is locked" errors; it won't really do anything to solve them.

`QuerySet.select_for_update()` not supported

SQLite does not support the `SELECT ... FOR UPDATE` syntax. Calling it will have no effect.

Parameters not quoted in `connection.queries`

`sqlite3` does not provide a way to retrieve the SQL after quoting and substituting the parameters. Instead, the SQL in `connection.queries` is rebuilt with a simple string interpolation. It may be incorrect. Make sure you add quotes where necessary before copying a query into a SQLite shell.

6.5.4 Oracle notes

Django supports Oracle Database Server versions 9i and higher. Oracle version 10g or later is required to use Django's `regex` and `iregex` query operators. You will also need at least version 4.3.1 of the cx_Oracle Python driver.

Note that due to a Unicode-corruption bug in `cx_Oracle` 5.0, that version of the driver should **not** be used with Django; `cx_Oracle` 5.0.1 resolved this issue, so if you'd like to use a more recent `cx_Oracle`, use version 5.0.1.

`cx_Oracle` 5.0.1 or greater can optionally be compiled with the `WITH_UNICODE` environment variable. This is recommended but not required.

In order for the `python manage.py syncdb` command to work, your Oracle database user must have privileges to run the following commands:

- CREATE TABLE
- CREATE SEQUENCE
- CREATE PROCEDURE
- CREATE TRIGGER

To run Django's test suite, the user needs these *additional* privileges:

- CREATE USER
- DROP USER
- CREATE TABLESPACE

- DROP TABLESPACE
- CONNECT WITH ADMIN OPTION
- RESOURCE WITH ADMIN OPTION

Connecting to the database

Your Django settings.py file should look something like this for Oracle:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': '',
        'PORT': '',
    }
}
```

If you don't use a `tnsnames.ora` file or a similar naming method that recognizes the SID ("xe" in this example), then fill in both `HOST` and `PORT` like so:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.oracle',
        'NAME': 'xe',
        'USER': 'a_user',
        'PASSWORD': 'a_password',
        'HOST': 'dbprod01ned.mycompany.com',
        'PORT': '1540',
    }
}
```

You should supply both `HOST` and `PORT`, or leave both as empty strings.

Threaded option

If you plan to run Django in a multithreaded environment (e.g. Apache in Windows using the default MPM module), then you **must** set the `threaded` option of your Oracle database configuration to True:

```
'OPTIONS': {
    'threaded': True,
},
```

Failure to do this may result in crashes and other odd behavior.

INSERT ... RETURNING INTO

By default, the Oracle backend uses a `RETURNING INTO` clause to efficiently retrieve the value of an `AutoField` when inserting new rows. This behavior may result in a `DatabaseError` in certain unusual setups, such as when inserting into a remote table, or into a view with an `INSTEAD OF` trigger. The `RETURNING INTO` clause can be disabled by setting the `use_returning_into` option of the database configuration to False:

```
'OPTIONS': {
    'use_returning_into': False,
},
```

In this case, the Oracle backend will use a separate `SELECT` query to retrieve AutoField values.

Naming issues

Oracle imposes a name length limit of 30 characters. To accommodate this, the backend truncates database identifiers to fit, replacing the final four characters of the truncated name with a repeatable MD5 hash value.

When running syncdb, an `ORA-06552` error may be encountered if certain Oracle keywords are used as the name of a model field or the value of a `db_column` option. Django quotes all identifiers used in queries to prevent most such problems, but this error can still occur when an Oracle datatype is used as a column name. In particular, take care to avoid using the names `date`, `timestamp`, `number` or `float` as a field name.

NULL and empty strings

Django generally prefers to use the empty string ('') rather than NULL, but Oracle treats both identically. To get around this, the Oracle backend ignores an explicit `null` option on fields that have the empty string as a possible value and generates DDL as if `null=True`. When fetching from the database, it is assumed that a `NULL` value in one of these fields really means the empty string, and the data is silently converted to reflect this assumption.

`TextField` limitations

The Oracle backend stores `TextFields` as `NCLOB` columns. Oracle imposes some limitations on the usage of such LOB columns in general:

- LOB columns may not be used as primary keys.
- LOB columns may not be used in indexes.
- LOB columns may not be used in a `SELECT DISTINCT` list. This means that attempting to use the `QuerySet.distinct` method on a model that includes `TextField` columns will result in an error when run against Oracle. As a workaround, use the `QuerySet.defer` method in conjunction with `distinct()` to prevent `TextField` columns from being included in the `SELECT DISTINCT` list.

6.5.5 Using a 3rd-party database backend

In addition to the officially supported databases, there are backends provided by 3rd parties that allow you to use other databases with Django:

- Sybase SQL Anywhere
- IBM DB2
- Microsoft SQL Server 2005
- Firebird
- ODBC
- ADSDB

The Django versions and ORM features supported by these unofficial backends vary considerably. Queries regarding the specific capabilities of these unofficial backends, along with any support queries, should be directed to the support channels provided by each 3rd party project.

6.6 django-admin.py and manage.py

`django-admin.py` is Django's command-line utility for administrative tasks. This document outlines all it can do.

In addition, `manage.py` is automatically created in each Django project. `manage.py` is a thin wrapper around `django-admin.py` that takes care of two things for you before delegating to `django-admin.py`:

- It puts your project's package on `sys.path`.
- It sets the `DJANGO_SETTINGS_MODULE` environment variable so that it points to your project's `settings.py` file.

The `django-admin.py` script should be on your system path if you installed Django via its `setup.py` utility. If it's not on your path, you can find it in `site-packages/django/bin` within your Python installation. Consider symlinking it from some place on your path, such as `/usr/local/bin`.

For Windows users, who do not have symlinking functionality available, you can copy `django-admin.py` to a location on your existing path or edit the `PATH` settings (under `Settings - Control Panel - System - Advanced - Environment...`) to point to its installed location.

Generally, when working on a single Django project, it's easier to use `manage.py`. Use `django-admin.py` with `DJANGO_SETTINGS_MODULE`, or the `--settings` command line option, if you need to switch between multiple Django settings files.

The command-line examples throughout this document use `django-admin.py` to be consistent, but any example can use `manage.py` just as well.

6.6.1 Usage

```
django-admin.py <command> [options]
manage.py <command> [options]
```

`command` should be one of the commands listed in this document. `options`, which is optional, should be zero or more of the options available for the given command.

Getting runtime help

`django-admin.py help`

Run `django-admin.py help` to display usage information and a list of the commands provided by each application.

Run `django-admin.py help --commands` to display a list of all available commands.

Run `django-admin.py help <command>` to display a description of the given command and a list of its available options.

App names

Many commands take a list of "app names." An "app name" is the basename of the package containing your models. For example, if your `INSTALLED_APPS` contains the string `'mysite.blog'`, the app name is `blog`.

Determining the version

django-admin.py version

Run `django-admin.py version` to display the current Django version.

The output follows the schema described in **PEP 386**:

```
1.4.dev17026
1.4a1
1.4
```

Displaying debug output

Use `--verbosity` to specify the amount of notification and debug information that `django-admin.py` should print to the console. For more details, see the documentation for the `--verbosity` option.

6.6.2 Available commands

cleanup

django-admin.py cleanup

Can be run as a cronjob or directly to clean out old data from the database (only expired sessions at the moment).

compilemessages

django-admin.py compilemessages

Compiles .po files created with `makemessages` to .mo files for use with the builtin gettext support. See *Internationalization and localization*.

Use the `--locale` option to specify the locale to process. If not provided, all locales are processed.

Example usage:

```
django-admin.py compilemessages --locale=pt_BR
```

createcachetable

django-admin.py createcachetable

Creates a cache table named `tablename` for use with the database cache backend. See *Django's cache framework* for more information.

The `--database` option can be used to specify the database onto which the cachetable will be installed.

dbshell

django-admin.py dbshell

Runs the command-line client for the database engine specified in your `ENGINE` setting, with the connection parameters specified in your `USER`, `PASSWORD`, etc., settings.

- For PostgreSQL, this runs the `psql` command-line client.

- For MySQL, this runs the `mysql` command-line client.
- For SQLite, this runs the `sqlite3` command-line client.

This command assumes the programs are on your `PATH` so that a simple call to the program name (`psql`, `mysql`, `sqlite3`) will find the program in the right place. There's no way to specify the location of the program manually.

The `--database` option can be used to specify the database onto which to open a shell.

diffsettings

`django-admin.py diffsettings`

Displays differences between the current settings file and Django's default settings.

Settings that don't appear in the defaults are followed by `"###"`. For example, the default settings don't define `ROOT_URLCONF`, so `ROOT_URLCONF` is followed by `"###"` in the output of `diffsettings`.

Note that Django's default settings live in `django/conf/global_settings.py`, if you're ever curious to see the full list of defaults.

dumpdata <appname appname appname.Model ...>

`django-admin.py dumpdata`

Outputs to standard output all data in the database associated with the named application(s).

If no application name is provided, all installed applications will be dumped.

The output of `dumpdata` can be used as input for `loaddata`.

Note that `dumpdata` uses the default manager on the model for selecting the records to dump. If you're using a *custom manager* as the default manager and it filters some of the available records, not all of the objects will be dumped. New in version 1.3: *Please see the release notes* The `--all` option may be provided to specify that `dumpdata` should use Django's base manager, dumping records which might otherwise be filtered or modified by a custom manager.

`-format` `<fmt>`

By default, `dumpdata` will format its output in JSON, but you can use the `--format` option to specify another format. Currently supported formats are listed in *Serialization formats*.

`-indent` `<num>`

By default, `dumpdata` will output all data on a single line. This isn't easy for humans to read, so you can use the `--indent` option to pretty-print the output with a number of indentation spaces.

The `--exclude` option may be provided to prevent specific applications from being dumped. New in version 1.3: *Please see the release notes* The `--exclude` option may also be provided to prevent specific models (specified as in the form of `appname.ModelName`) from being dumped.

In addition to specifying application names, you can provide a list of individual models, in the form of `appname.Model`. If you specify a model name to `dumpdata`, the dumped output will be restricted to that model, rather than the entire application. You can also mix application names and model names.

The `--database` option can be used to specify the database from which data will be dumped.

`-natural`

Use *natural keys* to represent any foreign key and many-to-many relationship with a model that provides a natural key definition. If you are dumping `contrib.auth` `Permission` objects or `contrib.contenttypes` `ContentType` objects, you should probably be using this flag.

flush

django-admin.py flush

Returns the database to the state it was in immediately after `syncdb` was executed. This means that all data will be removed from the database, any post-synchronization handlers will be re-executed, and the `initial_data` fixture will be re-installed.

The `--noinput` option may be provided to suppress all user prompts.

The `--database` option may be used to specify the database to flush.

–no-initial-data

New in version 1.5: *Please see the release notes* Use `--no-initial-data` to avoid loading the initial_data fixture.

inspectdb

django-admin.py inspectdb

Introspects the database tables in the database pointed-to by the `NAME` setting and outputs a Django model module (a `models.py` file) to standard output.

Use this if you have a legacy database with which you'd like to use Django. The script will inspect the database and create a model for each table within it.

As you might expect, the created models will have an attribute for every field in the table. Note that `inspectdb` has a few special cases in its field-name output:

- If `inspectdb` cannot map a column's type to a model field type, it'll use `TextField` and will insert the Python comment `'This field type is a guess.'` next to the field in the generated model.
- If the database column name is a Python reserved word (such as `'pass'`, `'class'` or `'for'`), `inspectdb` will append `'_field'` to the attribute name. For example, if a table has a column `'for'`, the generated model will have a field `'for_field'`, with the `db_column` attribute set to `'for'`. `inspectdb` will insert the Python comment `'Field renamed because it was a Python reserved word.'` next to the field.

This feature is meant as a shortcut, not as definitive model generation. After you run it, you'll want to look over the generated models yourself to make customizations. In particular, you'll need to rearrange models' order, so that models that refer to other models are ordered properly.

Primary keys are automatically introspected for PostgreSQL, MySQL and SQLite, in which case Django puts in the `primary_key=True` where needed.

`inspectdb` works with PostgreSQL, MySQL and SQLite. Foreign-key detection only works in PostgreSQL and with certain types of MySQL tables.

The `--database` option may be used to specify the database to introspect.

loaddata <fixture fixture ...>

django-admin.py loaddata

Searches for and loads the contents of the named fixture into the database.

The `--database` option can be used to specify the database onto which the data will be loaded.

What's a "fixture"?

A *fixture* is a collection of files that contain the serialized contents of the database. Each fixture has a unique name, and the files that comprise the fixture can be distributed over multiple directories, in multiple applications.

Django will search in three locations for fixtures:

1. In the `fixtures` directory of every installed application
2. In any directory named in the `FIXTURE_DIRS` setting
3. In the literal path named by the fixture

Django will load any and all fixtures it finds in these locations that match the provided fixture names.

If the named fixture has a file extension, only fixtures of that type will be loaded. For example:

```
django-admin.py loaddata mydata.json
```

would only load JSON fixtures called `mydata`. The fixture extension must correspond to the registered name of a *serializer* (e.g., `json` or `xml`).

If you omit the extensions, Django will search all available fixture types for a matching fixture. For example:

```
django-admin.py loaddata mydata
```

would look for any fixture of any fixture type called `mydata`. If a fixture directory contained `mydata.json`, that fixture would be loaded as a JSON fixture.

The fixtures that are named can include directory components. These directories will be included in the search path. For example:

```
django-admin.py loaddata foo/bar/mydata.json
```

would search `<appname>/fixtures/foo/bar/mydata.json` for each installed application, `<dirname>/foo/bar/mydata.json` for each directory in `FIXTURE_DIRS`, and the literal path `foo/bar/mydata.json`.

When fixture files are processed, the data is saved to the database as is. Model defined `save` methods and `pre_save` signals are not called.

Note that the order in which fixture files are processed is undefined. However, all fixture data is installed as a single transaction, so data in one fixture can reference data in another fixture. If the database backend supports row-level constraints, these constraints will be checked at the end of the transaction.

The `dumpdata` command can be used to generate input for `loaddata`.

Compressed fixtures

Fixtures may be compressed in `zip`, `gz`, or `bz2` format. For example:

```
django-admin.py loaddata mydata.json
```

would look for any of `mydata.json`, `mydata.json.zip`, `mydata.json.gz`, or `mydata.json.bz2`. The first file contained within a zip-compressed archive is used.

Note that if two fixtures with the same name but different fixture type are discovered (for example, if `mydata.json` and `mydata.xml.gz` were found in the same fixture directory), fixture installation will be aborted, and any data installed in the call to `loaddata` will be removed from the database.

MySQL with MyISAM and fixtures

The MyISAM storage engine of MySQL doesn't support transactions or constraints, so if you use MyISAM, you won't get validation of fixture data, or a rollback if multiple transaction files are found.

Database-specific fixtures

If you're in a multi-database setup, you might have fixture data that you want to load onto one database, but not onto another. In this situation, you can add database identifier into the names of your fixtures.

For example, if your `DATABASES` setting has a 'master' database defined, name the fixture `mydata.master.json` or `mydata.master.json.gz` and the fixture will only be loaded when you specify you want to load data into the `master` database.

makemessages

`django-admin.py makemessages`

Runs over the entire source tree of the current directory and pulls out all strings marked for translation. It creates (or updates) a message file in the conf/locale (in the django tree) or locale (for project and application) directory. After making changes to the messages files you need to compile them with `compilemessages` for use with the builtin gettext support. See the *i18n documentation* for details.

`-all`

Use the `--all` or `-a` option to update the message files for all available languages.

Example usage:

```
django-admin.py makemessages --all
```

`-extension`

Use the `--extension` or `-e` option to specify a list of file extensions to examine (default: ".html", ".txt").

Example usage:

```
django-admin.py makemessages --locale=de --extension xhtml
```

Separate multiple extensions with commas or use -e or –extension multiple times:

```
django-admin.py makemessages --locale=de --extension=html,txt --extension xml
```

Use the `--locale` option to specify the locale to process.

Example usage:

```
django-admin.py makemessages --locale=pt_BR
```

`-domain`

Use the `--domain` or `-d` option to change the domain of the messages files. Currently supported:

- `django` for all `*.py`, `*.html` and `*.txt` files (default)
- `djangojs` for `*.js` files

`-symlinks`

Use the `--symlinks` or `-s` option to follow symlinks to directories when looking for new translation strings.

Example usage:

```
django-admin.py makemessages --locale=de --symlinks
```

-ignore

Use the `--ignore` or `-i` option to ignore files or directories matching the given `glob`-style pattern. Use multiple times to ignore more.

These patterns are used by default: `'CVS'`, `'.*'`, `'*~'`

Example usage:

```
django-admin.py makemessages --locale=en_US --ignore=apps/* --ignore=secret/*.html
```

-no-default-ignore

Use the `--no-default-ignore` option to disable the default values of `--ignore`.

-no-wrap

New in version 1.3: *Please see the release notes* Use the `--no-wrap` option to disable breaking long message lines into several lines in language files.

-no-location

New in version 1.4: *Please see the release notes* Use the `--no-location` option to not write '`#: filename:line`' comment lines in language files. Note that using this option makes it harder for technically skilled translators to understand each message's context.

runfcgi [options]

django-admin.py runfcgi

Starts a set of FastCGI processes suitable for use with any Web server that supports the FastCGI protocol. See the *FastCGI deployment documentation* for details. Requires the Python FastCGI module from flup. New in version 1.4: Internally, this wraps the WSGI application object specified by the `WSGI_APPLICATION` setting. The options accepted by this command are passed to the FastCGI library and don't use the `'--'` prefix as is usual for other Django management commands.

protocol

```
protocol=PROTOCOL
```

Protocol to use. *PROTOCOL* can be `fcgi`, `scgi`, `ajp`, etc. (default is `fcgi`)

host

```
host=HOSTNAME
```

Hostname to listen on.

port

```
port=PORTNUM
```

Port to listen on.

socket

```
socket=FILE
```

UNIX socket to listen on.

method

`method=IMPL`

Possible values: `prefork` or `threaded` (default `prefork`)

maxrequests

`maxrequests=NUMBER`

Number of requests a child handles before it is killed and a new child is forked (0 means no limit).

maxspare

`maxspare=NUMBER`

Max number of spare processes / threads.

minspare

`minspare=NUMBER`

Min number of spare processes / threads.

maxchildren

`maxchildren=NUMBER`

Hard limit number of processes / threads.

daemonize

`daemonize=BOOL`

Whether to detach from terminal.

pidfile

`pidfile=FILE`

Write the spawned process-id to file *FILE*.

workdir

`workdir=DIRECTORY`

Change to directory *DIRECTORY* when daemonizing.

debug

`debug=BOOL`

Set to true to enable flup tracebacks.

outlog

`outlog=FILE`

Write stdout to the *FILE* file.

errlog

`errlog=FILE`

Write stderr to the *FILE* file.

umask

`umask=UMASK`

Umask to use when daemonizing. The value is interpeted as an octal number (default value is `022`).

Example usage:

```
django-admin.py runfcgi socket=/tmp/fcgi.sock method=prefork daemonize=true \
    pidfile=/var/run/django-fcgi.pid
```

Run a FastCGI server as a daemon and write the spawned PID in a file.

runserver [port or address:port]

django-admin.py runserver

Starts a lightweight development Web server on the local machine. By default, the server runs on port 8000 on the IP address `127.0.0.1`. You can pass in an IP address and port number explicitly.

If you run this script as a user with normal privileges (recommended), you might not have access to start a port on a low port number. Low port numbers are reserved for the superuser (root). New in version 1.4: This server uses the WSGI application object specified by the `WSGI_APPLICATION` setting. DO NOT USE THIS SERVER IN A PRODUCTION SETTING. It has not gone through security audits or performance tests. (And that's how it's gonna stay. We're in the business of making Web frameworks, not Web servers, so improving this server to be able to handle a production environment is outside the scope of Django.)

The development server automatically reloads Python code for each request, as needed. You don't need to restart the server for code changes to take effect.

When you start the server, and each time you change Python code while the server is running, the server will validate all of your installed models. (See the `validate` command below.) If the validator finds errors, it will print them to standard output, but it won't stop the server.

You can run as many servers as you want, as long as they're on separate ports. Just execute `django-admin.py runserver` more than once.

Note that the default IP address, `127.0.0.1`, is not accessible from other machines on your network. To make your development server viewable to other machines on the network, use its own IP address (e.g. `192.168.2.1`) or `0.0.0.0` or `::` (with IPv6 enabled). Changed in version 1.3: *Please see the release notes* You can provide an IPv6 address surrounded by brackets (e.g. `[200a::1]:8000`). This will automatically enable IPv6 support.

A hostname containing ASCII-only characters can also be used. Changed in version 1.3: *Please see the release notes* If the *staticfiles* contrib app is enabled (default in new projects) the `runserver` command will be overriden with an own `runserver` command.

-noreload

Use the `--noreload` option to disable the use of the auto-reloader. This means any Python code changes you make while the server is running will *not* take effect if the particular Python modules have already been loaded into memory.

Example usage:

```
django-admin.py runserver --noreload
```

-nothreading

New in version 1.4: *Please see the release notes* Since version 1.4, the development server is multithreaded by default. Use the `--nothreading` option to disable the use of threading in the development server.

-ipv6, -6

New in version 1.3: *Please see the release notes* Use the `--ipv6` (or shorter `-6`) option to tell Django to use IPv6 for the development server. This changes the default IP address from `127.0.0.1` to `::1`.

Example usage:

```
django-admin.py runserver --ipv6
```

Examples of using different ports and addresses

Port 8000 on IP address `127.0.0.1`:

```
django-admin.py runserver
```

Port 8000 on IP address `1.2.3.4`:

```
django-admin.py runserver 1.2.3.4:8000
```

Port 7000 on IP address `127.0.0.1`:

```
django-admin.py runserver 7000
```

Port 7000 on IP address `1.2.3.4`:

```
django-admin.py runserver 1.2.3.4:7000
```

Port 8000 on IPv6 address `::1`:

```
django-admin.py runserver -6
```

Port 7000 on IPv6 address `::1`:

```
django-admin.py runserver -6 7000
```

Port 7000 on IPv6 address `2001:0db8:1234:5678::9`:

```
django-admin.py runserver [2001:0db8:1234:5678::9]:7000
```

Port 8000 on IPv4 address of host `localhost`:

```
django-admin.py runserver localhost:8000
```

Port 8000 on IPv6 address of host `localhost`:

```
django-admin.py runserver -6 localhost:8000
```

Serving static files with the development server

By default, the development server doesn't serve any static files for your site (such as CSS files, images, things under `MEDIA_URL` and so forth). If you want to configure Django to serve static media, read *Managing static files*.

shell

`django-admin.py shell`

Starts the Python interactive interpreter.

Django will use IPython or bpython if either is installed. If you have a rich shell installed but want to force use of the "plain" Python interpreter, use the `--plain` option, like so:

```
django-admin.py shell --plain
```

Changed in version 1.5: *Please see the release notes* If you would like to specify either IPython or bpython as your interpreter if you have both installed you can specify an alternative interpreter interface with the `-i` or `--interface` options like so:

IPython:

```
django-admin.py shell -i ipython
django-admin.py shell --interface ipython
```

bpython:

```
django-admin.py shell -i bpython
django-admin.py shell --interface bpython
```

sql <appname appname ...>

django-admin.py sql

Prints the CREATE TABLE SQL statements for the given app name(s).

The `--database` option can be used to specify the database for which to print the SQL.

sqlall <appname appname ...>

django-admin.py sqlall

Prints the CREATE TABLE and initial-data SQL statements for the given app name(s).

Refer to the description of `sqlcustom` for an explanation of how to specify initial data.

The `--database` option can be used to specify the database for which to print the SQL.

sqlclear <appname appname ...>

django-admin.py sqlclear

Prints the DROP TABLE SQL statements for the given app name(s).

The `--database` option can be used to specify the database for which to print the SQL.

sqlcustom <appname appname ...>

django-admin.py sqlcustom

Prints the custom SQL statements for the given app name(s).

For each model in each specified app, this command looks for the file `<appname>/sql/<modelname>.sql`, where `<appname>` is the given app name and `<modelname>` is the model's name in lowercase. For example, if you have an app `news` that includes a `Story` model, `sqlcustom` will attempt to read a file `news/sql/story.sql` and append it to the output of this command.

Each of the SQL files, if given, is expected to contain valid SQL. The SQL files are piped directly into the database after all of the models' table-creation statements have been executed. Use this SQL hook to make any table modifications, or insert any SQL functions into the database.

Note that the order in which the SQL files are processed is undefined.

The `--database` option can be used to specify the database for which to print the SQL.

sqlflush

django-admin.py sqlflush

Prints the SQL statements that would be executed for the `flush` command.

The `--database` option can be used to specify the database for which to print the SQL.

sqlindexes <appname appname ...>

django-admin.py sqlindexes

Prints the CREATE INDEX SQL statements for the given app name(s).

The `--database` option can be used to specify the database for which to print the SQL.

sqlsequencereset <appname appname ...>

django-admin.py sqlsequencereset

Prints the SQL statements for resetting sequences for the given app name(s).

Sequences are indexes used by some database engines to track the next available number for automatically incremented fields.

Use this command to generate SQL which will fix cases where a sequence is out of sync with its automatically incremented field data.

The `--database` option can be used to specify the database for which to print the SQL.

startapp <appname> [destination]

django-admin.py startapp

Creates a Django app directory structure for the given app name in the current directory or the given destination. Changed in version 1.4: *Please see the release notes* By default the directory created contains a `models.py` file and other app template files. (See the source for more details.) If only the app name is given, the app directory will be created in the current working directory.

If the optional destination is provided, Django will use that existing directory rather than creating a new one. You can use '.' to denote the current working directory.

For example:

```
django-admin.py startapp myapp /Users/jezdez/Code/myapp
```

New in version 1.4: *Please see the release notes*

-template

With the `--template` option, you can use a custom app template by providing either the path to a directory with the app template file, or a path to a compressed file (`.tar.gz`, `.tar.bz2`, `.tgz`, `.tbz`, `.zip`) containing the app template files.

Django will also accept URLs (`http`, `https`, `ftp`) to compressed archives with the app template files, downloading and extracting them on the fly.

For example, this would look for an app template in the given directory when creating the `myapp` app:

```
django-admin.py startapp --template=/Users/jezdez/Code/my_app_template myapp
```

New in version 1.4: *Please see the release notes* When Django copies the app template files, it also renders certain files through the template engine: the files whose extensions match the `--extension` option (`py` by default) and the files whose names are passed with the `--name` option. The `template context` used is:

- Any option passed to the startapp command (among the command's supported options)
- `app_name` – the app name as passed to the command
- `app_directory` – the full path of the newly created app

Warning: When the app template files are rendered with the Django template engine (by default all `*.py` files), Django will also replace all stray template variables contained. For example, if one of the Python files contains a docstring explaining a particular feature related to template rendering, it might result in an incorrect example.
To work around this problem, you can use the `templatetag` templatetag to "escape" the various parts of the template syntax.

startproject <projectname> [destination]

`django-admin.py startproject`

Creates a Django project directory structure for the given project name in the current directory or the given destination. Changed in version 1.4: *Please see the release notes* By default, the new directory contains `manage.py` and a project package (containing a `settings.py` and other files). See the template source for details.

If only the project name is given, both the project directory and project package will be named `<projectname>` and the project directory will be created in the current working directory.

If the optional destination is provided, Django will use that existing directory as the project directory, and create `manage.py` and the project package within it. Use '.' to denote the current working directory.

For example:

```
django-admin.py startproject myproject /Users/jezdez/Code/myproject_repo
```

New in version 1.4: *Please see the release notes* As with the `startapp` command, the `--template` option lets you specify a directory, file path or URL of a custom project template. See the `startapp` documentation for details of supported project template formats.

For example, this would look for a project template in the given directory when creating the `myproject` project:

```
django-admin.py startproject --template=/Users/jezdez/Code/my_project_template myproject
```

When Django copies the project template files, it also renders certain files through the template engine: the files whose extensions match the `--extension` option (`py` by default) and the files whose names are passed with the `--name` option. The `template context` used is:

- Any option passed to the startproject command
- `project_name` – the project name as passed to the command
- `project_directory` – the full path of the newly created project
- `secret_key` – a random key for the `SECRET_KEY` setting

Please also see the *rendering warning* as mentioned for `startapp`.

syncdb

django-admin.py syncdb

Creates the database tables for all apps in `INSTALLED_APPS` whose tables have not already been created.

Use this command when you've added new applications to your project and want to install them in the database. This includes any apps shipped with Django that might be in `INSTALLED_APPS` by default. When you start a new project, run this command to install the default apps.

Syncdb will not alter existing tables

`syncdb` will only create tables for models which have not yet been installed. It will *never* issue `ALTER TABLE` statements to match changes made to a model class after installation. Changes to model classes and database schemas often involve some form of ambiguity and, in those cases, Django would have to guess at the correct changes to make. There is a risk that critical data would be lost in the process.

If you have made changes to a model and wish to alter the database tables to match, use the `sql` command to display the new SQL structure and compare that to your existing table schema to work out the changes.

If you're installing the `django.contrib.auth` application, `syncdb` will give you the option of creating a superuser immediately.

`syncdb` will also search for and install any fixture named `initial_data` with an appropriate extension (e.g. `json` or `xml`). See the documentation for `loaddata` for details on the specification of fixture data files.

The `--noinput` option may be provided to suppress all user prompts.

The `--database` option can be used to specify the database to synchronize.

–no-initial-data

New in version 1.5: *Please see the release notes* Use `--no-initial-data` to avoid loading the initial_data fixture.

test <app or test identifier>

django-admin.py test

Runs tests for all installed models. See *Testing Django applications* for more information.

-failfast

The `--failfast` option can be used to stop running tests and report the failure immediately after a test fails. New in version 1.4: *Please see the release notes*

-testrunner

The `--testrunner` option can be used to control the test runner class that is used to execute tests. If this value is provided, it overrides the value provided by the `TEST_RUNNER` setting. New in version 1.4: *Please see the release notes*

-liveserver

The `--liveserver` option can be used to override the default address where the live server (used with `LiveServerTestCase`) is expected to run from. The default value is `localhost:8081`.

testserver <fixture fixture ...>

django-admin.py testserver

Runs a Django development server (as in `runserver`) using data from the given fixture(s).

For example, this command:

```
django-admin.py testserver mydata.json
```

...would perform the following steps:

1. Create a test database, as described in *Testing Django applications*.
2. Populate the test database with fixture data from the given fixtures. (For more on fixtures, see the documentation for `loaddata` above.)
3. Runs the Django development server (as in `runserver`), pointed at this newly created test database instead of your production database.

This is useful in a number of ways:

- When you're writing *unit tests* of how your views act with certain fixture data, you can use `testserver` to interact with the views in a Web browser, manually.
- Let's say you're developing your Django application and have a "pristine" copy of a database that you'd like to interact with. You can dump your database to a fixture (using the `dumpdata` command, explained above), then use `testserver` to run your Web application with that data. With this arrangement, you have the flexibility of messing up your data in any way, knowing that whatever data changes you're making are only being made to a test database.

Note that this server does *not* automatically detect changes to your Python source code (as `runserver` does). It does, however, detect changes to templates.

-addrport `[port number or ipaddr:port]`

Use `--addrport` to specify a different port, or IP address and port, from the default of `127.0.0.1:8000`. This value follows exactly the same format and serves exactly the same function as the argument to the `runserver` command.

Examples:

To run the test server on port 7000 with `fixture1` and `fixture2`:

```
django-admin.py testserver --addrport 7000 fixture1 fixture2
django-admin.py testserver fixture1 fixture2 --addrport 7000
```

(The above statements are equivalent. We include both of them to demonstrate that it doesn't matter whether the options come before or after the fixture arguments.)

To run on 1.2.3.4:7000 with a `test` fixture:

```
django-admin.py testserver --addrport 1.2.3.4:7000 test
```

New in version 1.3: *Please see the release notes* The `--noinput` option may be provided to suppress all user prompts.

validate

django-admin.py validate

Validates all installed models (according to the `INSTALLED_APPS` setting) and prints validation errors to standard output.

6.6.3 Commands provided by applications

Some commands are only available when the `django.contrib` application that *implements* them has been `enabled`. This section describes them grouped by their application.

`django.contrib.auth`

changepassword

`django-admin.py changepassword`

This command is only available if Django's *authentication system* (`django.contrib.auth`) is installed.

Allows changing a user's password. It prompts you to enter twice the password of the user given as parameter. If they both match, the new password will be changed immediately. If you do not supply a user, the command will attempt to change the password whose username matches the current user. New in version 1.4: *Please see the release notes* Use the `--database` option to specify the database to query for the user. If it's not supplied, Django will use the `default` database.

Example usage:

```
django-admin.py changepassword ringo
```

createsuperuser

`django-admin.py createsuperuser`

This command is only available if Django's *authentication system* (`django.contrib.auth`) is installed.

Creates a superuser account (a user who has all permissions). This is useful if you need to create an initial superuser account but did not do so during `syncdb`, or if you need to programmatically generate superuser accounts for your site(s).

When run interactively, this command will prompt for a password for the new superuser account. When run non-interactively, no password will be set, and the superuser account will not be able to log in until a password has been manually set for it.

`--username`

`--email`

The username and email address for the new account can be supplied by using the `--username` and `--email` arguments on the command line. If either of those is not supplied, `createsuperuser` will prompt for it when running interactively. New in version 1.4: *Please see the release notes* Use the `--database` option to specify the database into which the superuser object will be saved.

`django.contrib.gis`

ogrinspect

This command is only available if *GeoDjango* (`django.contrib.gis`) is installed.

Please refer to its `description` in the GeoDjango documentation.

django.contrib.sitemaps

ping_google

This command is only available if the *Sitemaps framework* (`django.contrib.sitemaps`) is installed.

Please refer to its `description` in the Sitemaps documentation.

django.contrib.staticfiles

collectstatic

This command is only available if the *static files application* (`django.contrib.staticfiles`) is installed.

Please refer to its `description` in the *staticfiles* documentation.

findstatic

This command is only available if the *static files application* (`django.contrib.staticfiles`) is installed.

Please refer to its `description` in the *staticfiles* documentation.

6.6.4 Default options

Although some commands may allow their own custom options, every command allows for the following options:

`--pythonpath`

Example usage:

```
django-admin.py syncdb --pythonpath='/home/djangoprojects/myproject'
```

Adds the given filesystem path to the Python import search path. If this isn't provided, `django-admin.py` will use the `PYTHONPATH` environment variable.

Note that this option is unnecessary in `manage.py`, because it takes care of setting the Python path for you.

`--settings`

Example usage:

```
django-admin.py syncdb --settings=mysite.settings
```

Explicitly specifies the settings module to use. The settings module should be in Python package syntax, e.g. `mysite.settings`. If this isn't provided, `django-admin.py` will use the `DJANGO_SETTINGS_MODULE` environment variable.

Note that this option is unnecessary in `manage.py`, because it uses `settings.py` from the current project by default.

`--traceback`

Example usage:

```
django-admin.py syncdb --traceback
```

By default, `django-admin.py` will show a simple error message whenever an error occurs. If you specify `--traceback`, `django-admin.py` will output a full stack trace whenever an exception is raised.

--verbosity

Example usage:

```
django-admin.py syncdb --verbosity 2
```

Use `--verbosity` to specify the amount of notification and debug information that `django-admin.py` should print to the console.

- `0` means no output.
- `1` means normal output (default).
- `2` means verbose output.
- `3` means *very* verbose output.

6.6.5 Common options

The following options are not available on every command, but they are common to a number of commands.

--database

Used to specify the database on which a command will operate. If not specified, this option will default to an alias of `default`.

For example, to dump data from the database with the alias `master`:

```
django-admin.py dumpdata --database=master
```

--exclude

Exclude a specific application from the applications whose contents is output. For example, to specifically exclude the *auth* application from the output of dumpdata, you would call:

```
django-admin.py dumpdata --exclude=auth
```

If you want to exclude multiple applications, use multiple `--exclude` directives:

```
django-admin.py dumpdata --exclude=auth --exclude=contenttypes
```

--locale

Use the `--locale` or `-l` option to specify the locale to process. If not provided all locales are processed.

--noinput

Use the `--noinput` option to suppress all user prompting, such as "Are you sure?" confirmation messages. This is useful if `django-admin.py` is being executed as an unattended, automated script.

6.6.6 Extra niceties

Syntax coloring

The `django-admin.py` / `manage.py` commands will use pretty color-coded output if your terminal supports ANSI-colored output. It won't use the color codes if you're piping the command's output to another program.

The colors used for syntax highlighting can be customized. Django ships with three color palettes:

- `dark`, suited to terminals that show white text on a black background. This is the default palette.
- `light`, suited to terminals that show black text on a white background.

- nocolor, which disables syntax highlighting.

You select a palette by setting a DJANGO_COLORS environment variable to specify the palette you want to use. For example, to specify the light palette under a Unix or OS/X BASH shell, you would run the following at a command prompt:

```
export DJANGO_COLORS="light"
```

You can also customize the colors that are used. Django specifies a number of roles in which color is used:

- error - A major error.
- notice - A minor error.
- sql_field - The name of a model field in SQL.
- sql_coltype - The type of a model field in SQL.
- sql_keyword - A SQL keyword.
- sql_table - The name of a model in SQL.
- http_info - A 1XX HTTP Informational server response.
- http_success - A 2XX HTTP Success server response.
- http_not_modified - A 304 HTTP Not Modified server response.
- http_redirect - A 3XX HTTP Redirect server response other than 304.
- http_not_found - A 404 HTTP Not Found server response.
- http_bad_request - A 4XX HTTP Bad Request server response other than 404.
- http_server_error - A 5XX HTTP Server Error response.

Each of these roles can be assigned a specific foreground and background color, from the following list:

- black
- red
- green
- yellow
- blue
- magenta
- cyan
- white

Each of these colors can then be modified by using the following display options:

- bold
- underscore
- blink
- reverse
- conceal

A color specification follows one of the following patterns:

- role=fg

- `role=fg/bg`
- `role=fg,option,option`
- `role=fg/bg,option,option`

where `role` is the name of a valid color role, `fg` is the foreground color, `bg` is the background color and each `option` is one of the color modifying options. Multiple color specifications are then separated by semicolon. For example:

```
export DJANGO_COLORS="error=yellow/blue,blink;notice=magenta"
```

would specify that errors be displayed using blinking yellow on blue, and notices displayed using magenta. All other color roles would be left uncolored.

Colors can also be specified by extending a base palette. If you put a palette name in a color specification, all the colors implied by that palette will be loaded. So:

```
export DJANGO_COLORS="light;error=yellow/blue,blink;notice=magenta"
```

would specify the use of all the colors in the light color palette, *except* for the colors for errors and notices which would be overridden as specified.

Bash completion

If you use the Bash shell, consider installing the Django bash completion script, which lives in `extras/django_bash_completion` in the Django distribution. It enables tab-completion of `django-admin.py` and `manage.py` commands, so you can, for instance...

- Type `django-admin.py`.
- Press [TAB] to see all available options.
- Type `sql`, then [TAB], to see all available options whose names start with `sql`.

See *Writing custom django-admin commands* for how to add customized actions.

6.7 Running management commands from your code

`django.core.management.`**`call_command`**(*name*, **args*, ***options*)

To call a management command from code use `call_command`.

`name` the name of the command to call.

`*args` a list of arguments accepted by the command.

`**options` named options accepted on the command-line.

Examples:

```
from django.core import management
management.call_command('flush', verbosity=0, interactive=False)
management.call_command('loaddata', 'test_data', verbosity=0)
```

6.8 Django Exceptions

Django raises some Django specific exceptions as well as many standard Python exceptions.

6.8.1 Django-specific Exceptions

ObjectDoesNotExist and DoesNotExist

exception `DoesNotExist`

exception `ObjectDoesNotExist`

The `DoesNotExist` exception is raised when an object is not found for the given parameters of a query.

`ObjectDoesNotExist` is defined in `django.core.exceptions`. `DoesNotExist` is a subclass of the base `ObjectDoesNotExist` exception that is provided on every model class as a way of identifying the specific type of object that could not be found.

See `get()` for further information on `ObjectDoesNotExist` and `DoesNotExist`.

MultipleObjectsReturned

exception `MultipleObjectsReturned`

The `MultipleObjectsReturned` exception is raised by a query if only one object is expected, but multiple objects are returned. A base version of this exception is provided in `django.core.exceptions`; each model class contains a subclassed version that can be used to identify the specific object type that has returned multiple objects.

See `get()` for further information.

SuspiciousOperation

exception `SuspiciousOperation`

The `SuspiciousOperation` exception is raised when a user has performed an operation that should be considered suspicious from a security perspective, such as tampering with a session cookie.

PermissionDenied

exception `PermissionDenied`

The `PermissionDenied` exception is raised when a user does not have permission to perform the action requested.

ViewDoesNotExist

exception `ViewDoesNotExist`

The `ViewDoesNotExist` exception is raised by `django.core.urlresolvers` when a requested view does not exist.

MiddlewareNotUsed

exception `MiddlewareNotUsed`

The `MiddlewareNotUsed` exception is raised when a middleware is not used in the server configuration.

ImproperlyConfigured

exception `ImproperlyConfigured`
The `ImproperlyConfigured` exception is raised when Django is somehow improperly configured – for example, if a value in `settings.py` is incorrect or unparseable.

FieldError

exception `FieldError`
The `FieldError` exception is raised when there is a problem with a model field. This can happen for several reasons:

- A field in a model clashes with a field of the same name from an abstract base class
- An infinite loop is caused by ordering
- A keyword cannot be parsed from the filter parameters
- A field cannot be determined from a keyword in the query parameters
- A join is not permitted on the specified field
- A field name is invalid
- A query contains invalid order_by arguments

ValidationError

exception `ValidationError`
The `ValidationError` exception is raised when data fails form or model field validation. For more information about validation, see *Form and Field Validation*, *Model Field Validation* and the *Validator Reference*.

NoReverseMatch

exception `NoReverseMatch`
The `NoReverseMatch` exception is raised by `django.core.urlresolvers` when a matching URL in your URLconf cannot be identified based on the parameters supplied.

6.8.2 Database Exceptions

Django wraps the standard database exceptions `DatabaseError` and `IntegrityError` so that your Django code has a guaranteed common implementation of these classes. These database exceptions are provided in `django.db`.

exception `DatabaseError`

exception `IntegrityError`

The Django wrappers for database exceptions behave exactly the same as the underlying database exceptions. See **PEP 249**, the Python Database API Specification v2.0, for further information.

6.8.3 Transaction Exceptions

exception `TransactionManagementError`
The `TransactionManagementError` is raised for any and all problems related to database transactions. It is available from `django.db.transaction`.

6.8.4 Python Exceptions

Django raises built-in Python exceptions when appropriate as well. See the Python documentation for further information on the built-in `exceptions`.

6.9 File handling

6.9.1 The `File` object

The `django.core.files` module and its submodules contain built-in classes for basic file handling in Django.

The `File` Class

class `File` (*file_object*)

The `File` is a thin wrapper around Python's built-in file object with some Django-specific additions. Internally, Django uses this class any time it needs to represent a file.

`File` objects have the following attributes and methods:

`name`

The name of file including the relative path from `MEDIA_ROOT`.

`size`

The size of the file in bytes.

`file`

The underlying Python `file` object passed to `File`.

`mode`

The read/write mode for the file.

`open`(*[mode=None]*)

Open or reopen the file (which by definition also does `File.seek(0)`). The `mode` argument allows the same values as Python's standard `open()`.

When reopening a file, `mode` will override whatever mode the file was originally opened with; `None` means to reopen with the original mode.

`read`(*[num_bytes=None]*)

Read content from the file. The optional `size` is the number of bytes to read; if not specified, the file will be read to the end.

`__iter__`()

Iterate over the file yielding one line at a time.

`chunks`(*[chunk_size=None]*)

Iterate over the file yielding "chunks" of a given size. `chunk_size` defaults to 64 KB.

This is especially useful with very large files since it allows them to be streamed off disk and avoids storing the whole file in memory.

`multiple_chunks`(*[chunk_size=None]*)

Returns `True` if the file is large enough to require multiple chunks to access all of its content give some `chunk_size`.

`write`(*[content]*)

Writes the specified content string to the file. Depending on the storage system behind the scenes, this content might not be fully committed until `close()` is called on the file.

close()
Close the file.

In addition to the listed methods, `File` exposes the following attributes and methods of the underlying `file` object: `encoding`, `fileno`, `flush`, `isatty`, `newlines`, `read`, `readinto`, `readlines`, `seek`, `softspace`, `tell`, `truncate`, `writelines`, `xreadlines`.

The `ContentFile` Class

class ContentFile(*File*)
The `ContentFile` class inherits from `File`, but unlike `File` it operates on string content, rather than an actual file. For example:

```
from __future__ import unicode_literals
from django.core.files.base import ContentFile

f1 = ContentFile(b"my string content")
f2 = ContentFile("my unicode content encoded as UTF-8".encode('UTF-8'))
```

The `ImageFile` Class

class ImageFile(*file_object*)
Django provides a built-in class specifically for images. `django.core.files.images.ImageFile` inherits all the attributes and methods of `File`, and additionally provides the following:

width
Width of the image in pixels.

height
Height of the image in pixels.

Additional methods on files attached to objects

Any `File` that's associated with an object (as with `Car.photo`, below) will also have a couple of extra methods:

`File.`**save**(*name*, *content*[, *save=True*])
Saves a new file with the file name and contents provided. This will not replace the existing file, but will create a new file and update the object to point to it. If `save` is `True`, the model's `save()` method will be called once the file is saved. That is, these two lines:

```
>>> car.photo.save('myphoto.jpg', content, save=False)
>>> car.save()
```

are the same as this one line:

```
>>> car.photo.save('myphoto.jpg', content, save=True)
```

Note that the `content` argument must be an instance of either `File` or of a subclass of `File`, such as `ContentFile`.

`File.`**delete**([*save=True*])
Removes the file from the model instance and deletes the underlying file. If `save` is `True`, the model's `save()` method will be called once the file is deleted.

6.9.2 File storage API

Getting the current storage class

Django provides two convenient ways to access the current storage class:

class `DefaultStorage`

`DefaultStorage` provides lazy access to the current default storage system as defined by `DEFAULT_FILE_STORAGE`. `DefaultStorage` uses `get_storage_class()` internally.

`get_storage_class`(*[import_path=None]*)

Returns a class or module which implements the storage API.

When called without the `import_path` parameter `get_storage_class` will return the current default storage system as defined by `DEFAULT_FILE_STORAGE`. If `import_path` is provided, `get_storage_class` will attempt to import the class or module from the given path and will return it if successful. An exception will be raised if the import is unsuccessful.

The FileSystemStorage Class

class `FileSystemStorage`

The `FileSystemStorage` class implements basic file storage on a local filesystem. It inherits from `Storage` and provides implementations for all the public methods thereof.

Note: The `FileSystemStorage.delete` method will not raise raise an exception if the given file name does not exist.

The Storage Class

class `Storage`

The `Storage` class provides a standardized API for storing files, along with a set of default behaviors that all other storage systems can inherit or override as necessary.

`accessed_time`(*name*)

New in version 1.3: *Please see the release notes* Returns a `datetime` object containing the last accessed time of the file. For storage systems that aren't able to return the last accessed time this will raise `NotImplementedError` instead.

`created_time`(*name*)

New in version 1.3: *Please see the release notes* Returns a `datetime` object containing the creation time of the file. For storage systems that aren't able to return the creation time this will raise `NotImplementedError` instead.

`delete`(*name*)

Deletes the file referenced by `name`. If deletion is not supported on the targest storage system this will raise `NotImplementedError` instead

`exists`(*name*)

Returns `True` if a file referenced by the given name already exists in the storage system, or `False` if the name is available for a new file.

`get_available_name`(*name*)

Returns a filename based on the `name` parameter that's free and available for new content to be written to on the target storage system.

get_valid_name(*name*)
Returns a filename based on the `name` parameter that's suitable for use on the target storage system.

listdir(*path*)
Lists the contents of the specified path, returning a 2-tuple of lists; the first item being directories, the second item being files. For storage systems that aren't able to provide such a listing, this will raise a `NotImplementedError` instead.

modified_time(*name*)
New in version 1.3: *Please see the release notes* Returns a `datetime` object containing the last modified time. For storage systems that aren't able to return the last modified time, this will raise `NotImplementedError` instead.

open(*name*, *mode='rb'*)
Opens the file given by `name`. Note that although the returned file is guaranteed to be a `File` object, it might actually be some subclass. In the case of remote file storage this means that reading/writing could be quite slow, so be warned.

path(*name*)
The local filesystem path where the file can be opened using Python's standard `open()`. For storage systems that aren't accessible from the local filesystem, this will raise `NotImplementedError` instead.

save(*name*, *content*)
Saves a new file using the storage system, preferably with the name specified. If there already exists a file with this name `name`, the storage system may modify the filename as necessary to get a unique name. The actual name of the stored file will be returned.

The `content` argument must be an instance of `django.core.files.File` or of a subclass of `File`.

size(*name*)
Returns the total size, in bytes, of the file referenced by `name`. For storage systems that aren't able to return the file size this will raise `NotImplementedError` instead.

url(*name*)
Returns the URL where the contents of the file referenced by `name` can be accessed. For storage systems that don't support access by URL this will raise `NotImplementedError` instead.

6.10 Forms

Detailed form API reference. For introductory material, see *Working with forms*.

6.10.1 The Forms API

About this document

This document covers the gritty details of Django's forms API. You should read the *introduction to working with forms* first.

Bound and unbound forms

A `Form` instance is either **bound** to a set of data, or **unbound**.

- If it's **bound** to a set of data, it's capable of validating that data and rendering the form as HTML with the data displayed in the HTML.

- If it's **unbound**, it cannot do validation (because there's no data to validate!), but it can still render the blank form as HTML.

class Form

To create an unbound `Form` instance, simply instantiate the class:

```
>>> f = ContactForm()
```

To bind data to a form, pass the data as a dictionary as the first parameter to your `Form` class constructor:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
```

In this dictionary, the keys are the field names, which correspond to the attributes in your `Form` class. The values are the data you're trying to validate. These will usually be strings, but there's no requirement that they be strings; the type of data you pass depends on the `Field`, as we'll see in a moment.

`Form.is_bound`

If you need to distinguish between bound and unbound form instances at runtime, check the value of the form's `is_bound` attribute:

```
>>> f = ContactForm()
>>> f.is_bound
False
>>> f = ContactForm({'subject': 'hello'})
>>> f.is_bound
True
```

Note that passing an empty dictionary creates a *bound* form with empty data:

```
>>> f = ContactForm({})
>>> f.is_bound
True
```

If you have a bound `Form` instance and want to change the data somehow, or if you want to bind an unbound `Form` instance to some data, create another `Form` instance. There is no way to change data in a `Form` instance. Once a `Form` instance has been created, you should consider its data immutable, whether it has data or not.

Using forms to validate data

`Form.is_valid()`

The primary task of a `Form` object is to validate data. With a bound `Form` instance, call the `is_valid()` method to run validation and return a boolean designating whether the data was valid:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
```

Let's try with some invalid data. In this case, `subject` is blank (an error, because all fields are required by default) and `sender` is not a valid email address:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
```

`Form.errors`

Access the `errors` attribute to get a dictionary of error messages:

```
>>> f.errors
{'sender': [u'Enter a valid e-mail address.'], 'subject': [u'This field is required.']}
```

In this dictionary, the keys are the field names, and the values are lists of Unicode strings representing the error messages. The error messages are stored in lists because a field can have multiple error messages.

You can access `errors` without having to call `is_valid()` first. The form's data will be validated the first time either you call `is_valid()` or access `errors`.

The validation routines will only get called once, regardless of how many times you access `errors` or call `is_valid()`. This means that if validation has side effects, those side effects will only be triggered once.

Behavior of unbound forms

It's meaningless to validate a form with no data, but, for the record, here's what happens with unbound forms:

```
>>> f = ContactForm()
>>> f.is_valid()
False
>>> f.errors
{}
```

Dynamic initial values

`Form.initial`

Use `initial` to declare the initial value of form fields at runtime. For example, you might want to fill in a `username` field with the username of the current session.

To accomplish this, use the `initial` argument to a `Form`. This argument, if given, should be a dictionary mapping field names to initial values. Only include the fields for which you're specifying an initial value; it's not necessary to include every field in your form. For example:

```
>>> f = ContactForm(initial={'subject': 'Hi there!'})
```

These values are only displayed for unbound forms, and they're not used as fallback values if a particular value isn't provided.

Note that if a `Field` defines `initial` *and* you include `initial` when instantiating the `Form`, then the latter `initial` will have precedence. In this example, `initial` is provided both at the field level and at the form instance level, and the latter gets precedence:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='class')
...     url = forms.URLField()
...     comment = forms.CharField()
```

```
>>> f = CommentForm(initial={'name': 'instance'}, auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="instance" /></td></tr>
<tr><th>Url:</th><td><input type="text" name="url" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

Accessing "clean" data

`Form.cleaned_data`

Each field in a `Form` class is responsible not only for validating data, but also for "cleaning" it – normalizing it to a consistent format. This is a nice feature, because it allows data for a particular field to be input in a variety of ways, always resulting in consistent output.

For example, `DateField` normalizes input into a Python `datetime.date` object. Regardless of whether you pass it a string in the format `'1994-07-15'`, a `datetime.date` object, or a number of other formats, `DateField` will always normalize it to a `datetime.date` object as long as it's valid.

Once you've created a `Form` instance with a set of data and validated it, you can access the clean data via its `cleaned_data` attribute:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'cc_myself': True, 'message': u'Hi there', 'sender': u'foo@example.com', 'subject': u'hello'}
```

Note that any text-based field – such as `CharField` or `EmailField` – always cleans the input into a Unicode string. We'll cover the encoding implications later in this document.

If your data does *not* validate, your `Form` instance will not have a `cleaned_data` attribute:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
>>> f.cleaned_data
Traceback (most recent call last):
...
AttributeError: 'ContactForm' object has no attribute 'cleaned_data'
```

`cleaned_data` will always *only* contain a key for fields defined in the `Form`, even if you pass extra data when you define the `Form`. In this example, we pass a bunch of extra fields to the `ContactForm` constructor, but `cleaned_data` contains only the form's fields:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True,
...         'extra_field_1': 'foo',
...         'extra_field_2': 'bar',
```

```
...         'extra_field_3': 'baz'}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data # Doesn't contain extra_field_1, etc.
{'cc_myself': True, 'message': u'Hi there', 'sender': u'foo@example.com', 'subject': u'hello'}
```

`cleaned_data` will include a key and value for *all* fields defined in the `Form`, even if the data didn't include a value for fields that are not required. In this example, the data dictionary doesn't include a value for the `nick_name` field, but `cleaned_data` includes it, with an empty value:

```
>>> class OptionalPersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
...     nick_name = CharField(required=False)
>>> data = {'first_name': u'John', 'last_name': u'Lennon'}
>>> f = OptionalPersonForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'nick_name': u'', 'first_name': u'John', 'last_name': u'Lennon'}
```

In this above example, the `cleaned_data` value for `nick_name` is set to an empty string, because `nick_name` is `CharField`, and `CharField`s treat empty values as an empty string. Each field type knows what its "blank" value is – e.g., for `DateField`, it's `None` instead of the empty string. For full details on each field's behavior in this case, see the "Empty value" note for each field in the "Built-in `Field` classes" section below.

You can write code to perform validation for particular form fields (based on their name) or for the form as a whole (considering combinations of various fields). More information about this is in *Form and field validation*.

Outputting forms as HTML

The second task of a `Form` object is to render itself as HTML. To do so, simply `print` it:

```
>>> f = ContactForm()
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name="sub
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_mes
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name="sender" id="id_sender
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself"
```

If the form is bound to data, the HTML output will include that data appropriately. For example, if a field is represented by an `<input type="text">`, the data will be in the `value` attribute. If a field is represented by an `<input type="checkbox">`, then that HTML will include `checked="checked"` if appropriate:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> print(f)
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name="sub
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_mes
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name="sender" id="id_sender
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself"
```

This default output is a two-column HTML table, with a `<tr>` for each field. Notice the following:

- For flexibility, the output does *not* include the <table> and </table> tags, nor does it include the <form> and </form> tags or an <input type="submit"> tag. It's your job to do that.
- Each field type has a default HTML representation. CharField and EmailField are represented by an <input type="text">. BooleanField is represented by an <input type="checkbox">. Note these are merely sensible defaults; you can specify which HTML to use for a given field by using widgets, which we'll explain shortly.
- The HTML name for each tag is taken directly from its attribute name in the ContactForm class.
- The text label for each field – e.g. 'Subject:', 'Message:' and 'Cc myself:' is generated from the field name by converting all underscores to spaces and upper-casing the first letter. Again, note these are merely sensible defaults; you can also specify labels manually.
- Each text label is surrounded in an HTML <label> tag, which points to the appropriate form field via its id. Its id, in turn, is generated by prepending 'id_' to the field name. The id attributes and <label> tags are included in the output by default, to follow best practices, but you can change that behavior.

Although <table> output is the default output style when you print a form, other output styles are available. Each style is available as a method on a form object, and each rendering method returns a Unicode object.

as_p()

Form.**as_p**()

as_p() renders the form as a series of <p> tags, with each <p> containing one field:

```
>>> f = ContactForm()
>>> f.as_p()
u'<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject"
>>> print(f.as_p())
<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" ma
<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" />
<p><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_c
```

as_ul()

Form.**as_ul**()

as_ul() renders the form as a series of <li> tags, with each <li> containing one field. It does *not* include the <ul> or , so that you can specify any HTML attributes on the <ul> for flexibility:

```
>>> f = ContactForm()
>>> f.as_ul()
u'<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject"
>>> print(f.as_ul())
<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" m
<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /
<li><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></l
<li><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_
```

as_table()

Form.**as_table**()

Finally, as_table() outputs the form as an HTML <table>. This is exactly the same as print. In fact, when you print a form object, it calls its as_table() method behind the scenes:

```
>>> f = ContactForm()
>>> f.as_table()
u'<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" na
>>> print(f.as_table())
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="i
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name="sender" id="id_s
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_mys
```

Styling required or erroneous form rows

It's pretty common to style form rows and fields that are required or have errors. For example, you might want to present required form rows in bold and highlight errors in red.

The `Form` class has a couple of hooks you can use to add `class` attributes to required rows or to rows with errors: simply set the `Form.error_css_class` and/or `Form.required_css_class` attributes:

```
class ContactForm(Form):
    error_css_class = 'error'
    required_css_class = 'required'

    # ... and the rest of your fields here
```

Once you've done that, rows will be given `"error"` and/or `"required"` classes, as needed. The HTML will look something like:

```
>>> f = ContactForm(data)
>>> print(f.as_table())
<tr class="required"><th><label for="id_subject">Subject:</label>    ...
<tr class="required"><th><label for="id_message">Message:</label>    ...
<tr class="required error"><th><label for="id_sender">Sender:</label>      ...
<tr><th><label for="id_cc_myself">Cc myself:<label> ...
```

Configuring HTML `<label>` tags

An HTML `<label>` tag designates which label text is associated with which form element. This small enhancement makes forms more usable and more accessible to assistive devices. It's always a good idea to use `<label>` tags.

By default, the form rendering methods include HTML `id` attributes on the form elements and corresponding `<label>` tags around the labels. The `id` attribute values are generated by prepending `id_` to the form field names. This behavior is configurable, though, if you want to change the `id` convention or remove HTML `id` attributes and `<label>` tags entirely.

Use the `auto_id` argument to the `Form` constructor to control the label and `id` behavior. This argument must be `True`, `False` or a string.

If `auto_id` is `False`, then the form output will not include `<label>` tags nor `id` attributes:

```
>>> f = ContactForm(auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" /></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" /></td></tr>
<tr><th>Sender:</th><td><input type="text" name="sender" /></td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" /></li>
```

```
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="text" name="sender" /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="text" name="sender" /></p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

If `auto_id` is set to `True`, then the form output *will* include `<label>` tags and will simply use the field name as its `id` for each form field:

```
>>> f = ContactForm(auto_id=True)
>>> print(f.as_table())
<tr><th><label for="subject">Subject:</label></th><td><input id="subject" type="text" name="subject"
<tr><th><label for="message">Message:</label></th><td><input type="text" name="message" id="message"
<tr><th><label for="sender">Sender:</label></th><td><input type="text" name="sender" id="sender" /></
<tr><th><label for="cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id=
>>> print(f.as_ul())
<li><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="1
<li><label for="message">Message:</label> <input type="text" name="message" id="message" /></li>
<li><label for="sender">Sender:</label> <input type="text" name="sender" id="sender" /></li>
<li><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="cc_myself"
>>> print(f.as_p())
<p><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="10
<p><label for="message">Message:</label> <input type="text" name="message" id="message" /></p>
<p><label for="sender">Sender:</label> <input type="text" name="sender" id="sender" /></p>
<p><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="cc_myself" /
```

If `auto_id` is set to a string containing the format character `'%s'`, then the form output will include `<label>` tags, and will generate `id` attributes based on the format string. For example, for a format string `'field_%s'`, a field named `subject` will get the `id` value `'field_subject'`. Continuing our example:

```
>>> f = ContactForm(auto_id='id_for_%s')
>>> print(f.as_table())
<tr><th><label for="id_for_subject">Subject:</label></th><td><input id="id_for_subject" type="text" n
<tr><th><label for="id_for_message">Message:</label></th><td><input type="text" name="message" id="id
<tr><th><label for="id_for_sender">Sender:</label></th><td><input type="text" name="sender" id="id_fo
<tr><th><label for="id_for_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myse
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" name="subject
<li><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_for_message
<li><label for="id_for_sender">Sender:</label> <input type="text" name="sender" id="id_for_sender" />
<li><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_f
>>> print(f.as_p())
<p><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" name="subject"
<p><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_for_message"
<p><label for="id_for_sender">Sender:</label> <input type="text" name="sender" id="id_for_sender" /><
<p><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_fo
```

If `auto_id` is set to any other true value – such as a string that doesn't include `%s` – then the library will act as if `auto_id` is `True`.

By default, `auto_id` is set to the string `'id_%s'`.

Normally, a colon (`:`) will be appended after any label name when a form is rendered. It's possible to change the colon to another character, or omit it entirely, using the `label_suffix` parameter:

```
>>> f = ContactForm(auto_id='id_for_%s', label_suffix='')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject</label> <input id="id_for_subject" type="text" name="subject"
<li><label for="id_for_message">Message</label> <input type="text" name="message" id="id_for_message"
<li><label for="id_for_sender">Sender</label> <input type="text" name="sender" id="id_for_sender" /><
<li><label for="id_for_cc_myself">Cc myself</label> <input type="checkbox" name="cc_myself" id="id_fo
>>> f = ContactForm(auto_id='id_for_%s', label_suffix=' ->')
>>> print(f.as_ul())
<li><label for="id_for_subject">Subject -></label> <input id="id_for_subject" type="text" name="subje
<li><label for="id_for_message">Message -></label> <input type="text" name="message" id="id_for_messa
<li><label for="id_for_sender">Sender -></label> <input type="text" name="sender" id="id_for_sender"
<li><label for="id_for_cc_myself">Cc myself -></label> <input type="checkbox" name="cc_myself" id="id
```

Note that the label suffix is added only if the last character of the label isn't a punctuation character (`.`, `!`, `?` or `:`)

Notes on field ordering

In the `as_p()`, `as_ul()` and `as_table()` shortcuts, the fields are displayed in the order in which you define them in your form class. For example, in the `ContactForm` example, the fields are defined in the order `subject`, `message`, `sender`, `cc_myself`. To reorder the HTML output, just change the order in which those fields are listed in the class.

How errors are displayed

If you render a bound `Form` object, the act of rendering will automatically run the form's validation if it hasn't already happened, and the HTML output will include the validation errors as a `<ul class="errorlist">` near the field. The particular positioning of the error messages depends on the output method you're using:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid email address',
...         'cc_myself': True}
>>> f = ContactForm(data, auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><ul class="errorlist"><li>This field is required.</li></ul><input type="text
<tr><th>Message:</th><td><input type="text" name="message" value="Hi there" /></td></tr>
<tr><th>Sender:</th><td><ul class="errorlist"><li>Enter a valid e-mail address.</li></ul><input type=
<tr><th>Cc myself:</th><td><input checked="checked" type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul())
<li><ul class="errorlist"><li>This field is required.</li></ul>Subject: <input type="text" name="subj
<li>Message: <input type="text" name="message" value="Hi there" /></li>
<li><ul class="errorlist"><li>Enter a valid e-mail address.</li></ul>Sender: <input type="text" name=
<li>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p><ul class="errorlist"><li>This field is required.</li></ul></p>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
<p><ul class="errorlist"><li>Enter a valid e-mail address.</li></ul></p>
<p>Sender: <input type="text" name="sender" value="invalid email address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

Customizing the error list format

By default, forms use `django.forms.util.ErrorList` to format validation errors. If you'd like to use an alternate class for displaying errors, you can pass that in at construction time:

```
>>> from django.forms.util import ErrorList
>>> class DivErrorList(ErrorList):
...     def __unicode__(self):
...         return self.as_divs()
...     def as_divs(self):
...         if not self: return u''
...         return u'<div class="errorlist">%s</div>' % ''.join([u'<div class="error">%s</div>' % e f
>>> f = ContactForm(data, auto_id=False, error_class=DivErrorList)
>>> f.as_p()
<div class="errorlist"><div class="error">This field is required.</div></div>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
<div class="errorlist"><div class="error">Enter a valid e-mail address.</div></div>
<p>Sender: <input type="text" name="sender" value="invalid email address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

More granular output

The `as_p()`, `as_ul()` and `as_table()` methods are simply shortcuts for lazy developers – they're not the only way a form object can be displayed.

class BoundField

Used to display HTML or access attributes for a single field of a `Form` instance.

The `__unicode__()` and `__str__()` methods of this object displays the HTML for this field.

To retrieve a single `BoundField`, use dictionary lookup syntax on your form using the field's name as the key:

```
>>> form = ContactForm()
>>> print(form['subject'])
<input id="id_subject" type="text" name="subject" maxlength="100" />
```

To retrieve all `BoundField` objects, iterate the form:

```
>>> form = ContactForm()
>>> for boundfield in form: print(boundfield)
<input id="id_subject" type="text" name="subject" maxlength="100" />
<input type="text" name="message" id="id_message" />
<input type="text" name="sender" id="id_sender" />
<input type="checkbox" name="cc_myself" id="id_cc_myself" />
```

The field-specific output honors the form object's `auto_id` setting:

```
>>> f = ContactForm(auto_id=False)
>>> print(f['message'])
<input type="text" name="message" />
>>> f = ContactForm(auto_id='id_%s')
>>> print(f['message'])
<input type="text" name="message" id="id_message" />
```

For a field's list of errors, access the field's `errors` attribute.

`BoundField.`**errors**

A list-like object that is displayed as an HTML `<ul class="errorlist">` when printed:

```
>>> data = {'subject': 'hi', 'message': '', 'sender': '', 'cc_myself': ''}
>>> f = ContactForm(data, auto_id=False)
>>> print(f['message'])
<input type="text" name="message" />
>>> f['message'].errors
[u'This field is required.']
>>> print(f['message'].errors)
<ul class="errorlist"><li>This field is required.</li></ul>
>>> f['subject'].errors
[]
>>> print(f['subject'].errors)

>>> str(f['subject'].errors)
''
```

`BoundField.`**`css_classes`**`()`

When you use Django's rendering shortcuts, CSS classes are used to indicate required form fields or fields that contain errors. If you're manually rendering a form, you can access these CSS classes using the `css_classes` method:

```
>>> f = ContactForm(data)
>>> f['message'].css_classes()
'required'
```

If you want to provide some additional classes in addition to the error and required classes that may be required, you can provide those classes as an argument:

```
>>> f = ContactForm(data)
>>> f['message'].css_classes('foo bar')
'foo bar required'
```

`BoundField.`**`value`**`()`

New in version 1.3: *Please see the release notes*

Use this method to render the raw value of this field as it would be rendered by a `Widget`:

```
>>> initial = {'subject': 'welcome'}
>>> unbound_form = ContactForm(initial=initial)
>>> bound_form = ContactForm(data, initial=initial)
>>> print(unbound_form['subject'].value())
welcome
>>> print(bound_form['subject'].value())
hi
```

Binding uploaded files to a form

Dealing with forms that have `FileField` and `ImageField` fields is a little more complicated than a normal form.

Firstly, in order to upload files, you'll need to make sure that your `<form>` element correctly defines the `enctype` as `"multipart/form-data"`:

```
<form enctype="multipart/form-data" method="post" action="/foo/">
```

Secondly, when you use the form, you need to bind the file data. File data is handled separately to normal form data, so when your form contains a `FileField` and `ImageField`, you will need to specify a second argument when you bind your form. So if we extend our ContactForm to include an `ImageField` called `mugshot`, we need to bind the file data containing the mugshot image:

```
# Bound form with an image field
>>> from django.core.files.uploadedfile import SimpleUploadedFile
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> file_data = {'mugshot': SimpleUploadedFile('face.jpg', <file data>)}
>>> f = ContactFormWithMugshot(data, file_data)
```

In practice, you will usually specify `request.FILES` as the source of file data (just like you use `request.POST` as the source of form data):

```
# Bound form with an image field, data from the request
>>> f = ContactFormWithMugshot(request.POST, request.FILES)
```

Constructing an unbound form is the same as always – just omit both form data *and* file data:

```
# Unbound form with a image field
>>> f = ContactFormWithMugshot()
```

Testing for multipart forms

If you're writing reusable views or templates, you may not know ahead of time whether your form is a multipart form or not. The `is_multipart()` method tells you whether the form requires multipart encoding for submission:

```
>>> f = ContactFormWithMugshot()
>>> f.is_multipart()
True
```

Here's an example of how you might use this in a template:

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post" action="/foo/">
{% else %}
    <form method="post" action="/foo/">
{% endif %}
{{ form }}
</form>
```

Subclassing forms

If you have multiple `Form` classes that share fields, you can use subclassing to remove redundancy.

When you subclass a custom `Form` class, the resulting subclass will include all fields of the parent class(es), followed by the fields you define in the subclass.

In this example, `ContactFormWithPriority` contains all the fields from `ContactForm`, plus an additional field, `priority`. The `ContactForm` fields are ordered first:

```
>>> class ContactFormWithPriority(ContactForm):
...     priority = forms.CharField()
>>> f = ContactFormWithPriority(auto_id=False)
>>> print(f.as_ul())
<li>Subject: <input type="text" name="subject" maxlength="100" /></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="text" name="sender" /></li>
```

```
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
<li>Priority: <input type="text" name="priority" /></li>
```

It's possible to subclass multiple forms, treating forms as "mix-ins." In this example, `BeatleForm` subclasses both `PersonForm` and `InstrumentForm` (in that order), and its field list includes the fields from the parent classes:

```
>>> class PersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
>>> class InstrumentForm(Form):
...     instrument = CharField()
>>> class BeatleForm(PersonForm, InstrumentForm):
...     haircut_type = CharField()
>>> b = BeatleForm(auto_id=False)
>>> print(b.as_ul())
<li>First name: <input type="text" name="first_name" /></li>
<li>Last name: <input type="text" name="last_name" /></li>
<li>Instrument: <input type="text" name="instrument" /></li>
<li>Haircut type: <input type="text" name="haircut_type" /></li>
```

Prefixes for forms

`Form.`**`prefix`**

You can put several Django forms inside one `<form>` tag. To give each `Form` its own namespace, use the `prefix` keyword argument:

```
>>> mother = PersonForm(prefix="mother")
>>> father = PersonForm(prefix="father")
>>> print(mother.as_ul())
<li><label for="id_mother-first_name">First name:</label> <input type="text" name="mother-first_name"
<li><label for="id_mother-last_name">Last name:</label> <input type="text" name="mother-last_name" id
>>> print(father.as_ul())
<li><label for="id_father-first_name">First name:</label> <input type="text" name="father-first_name"
<li><label for="id_father-last_name">Last name:</label> <input type="text" name="father-last_name" id
```

6.10.2 Form fields

class `Field`(***kwargs*)

When you create a `Form` class, the most important part is defining the fields of the form. Each field has custom validation logic, along with a few other hooks.

`Field.`**`clean`**(*value*)

Although the primary way you'll use `Field` classes is in `Form` classes, you can also instantiate them and use them directly to get a better idea of how they work. Each `Field` instance has a `clean()` method, which takes a single argument and either raises a `django.forms.ValidationError` exception or returns the clean value:

```
>>> from django import forms
>>> f = forms.EmailField()
>>> f.clean('foo@example.com')
u'foo@example.com'
>>> f.clean('invalid email address')
Traceback (most recent call last):
...
ValidationError: [u'Enter a valid e-mail address.']
```

Core field arguments

Each `Field` class constructor takes at least these arguments. Some `Field` classes take additional, field-specific arguments, but the following should *always* be accepted:

required

`Field.`**`required`**

By default, each `Field` class assumes the value is required, so if you pass an empty value – either `None` or the empty string (`""`) – then `clean()` will raise a `ValidationError` exception:

```
>>> f = forms.CharField()
>>> f.clean('foo')
u'foo'
>>> f.clean('')
Traceback (most recent call last):
...
ValidationError: [u'This field is required.']
>>> f.clean(None)
Traceback (most recent call last):
...
ValidationError: [u'This field is required.']
>>> f.clean(' ')
u' '
>>> f.clean(0)
u'0'
>>> f.clean(True)
u'True'
>>> f.clean(False)
u'False'
```

To specify that a field is *not* required, pass `required=False` to the `Field` constructor:

```
>>> f = forms.CharField(required=False)
>>> f.clean('foo')
u'foo'
>>> f.clean('')
u''
>>> f.clean(None)
u''
>>> f.clean(0)
u'0'
>>> f.clean(True)
u'True'
>>> f.clean(False)
u'False'
```

If a `Field` has `required=False` and you pass `clean()` an empty value, then `clean()` will return a *normalized* empty value rather than raising `ValidationError`. For `CharField`, this will be a Unicode empty string. For other `Field` classes, it might be `None`. (This varies from field to field.)

label

`Field.`**`label`**

The `label` argument lets you specify the "human-friendly" label for this field. This is used when the `Field` is displayed in a `Form`.

As explained in "Outputting forms as HTML" above, the default label for a `Field` is generated from the field name by converting all underscores to spaces and upper-casing the first letter. Specify `label` if that default behavior doesn't result in an adequate label.

Here's a full example `Form` that implements `label` for two of its fields. We've specified `auto_id=False` to simplify the output:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(label='Your name')
...     url = forms.URLField(label='Your Web site', required=False)
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Your name:</th><td><input type="text" name="name" /></td></tr>
<tr><th>Your Web site:</th><td><input type="text" name="url" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

initial

`Field.`**`initial`**

The `initial` argument lets you specify the initial value to use when rendering this `Field` in an unbound `Form`.

To specify dynamic initial data, see the `Form.initial` parameter.

The use-case for this is when you want to display an "empty" form in which a field is initialized to a particular value. For example:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" /></td></tr>
<tr><th>Url:</th><td><input type="text" name="url" value="http://" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

You may be thinking, why not just pass a dictionary of the initial values as data when displaying the form? Well, if you do that, you'll trigger validation, and the HTML output will include any validation errors:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField()
...     url = forms.URLField()
...     comment = forms.CharField()
>>> default_data = {'name': 'Your name', 'url': 'http://'}
>>> f = CommentForm(default_data, auto_id=False)
>>> print(f)
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" /></td></tr>
<tr><th>Url:</th><td><ul class="errorlist"><li>Enter a valid URL.</li></ul><input type="text" name="u
<tr><th>Comment:</th><td><ul class="errorlist"><li>This field is required.</li></ul><input type="text
```

This is why `initial` values are only displayed for unbound forms. For bound forms, the HTML output will use the bound data.

Also note that `initial` values are *not* used as "fallback" data in validation if a particular field's value is not given. `initial` values are *only* intended for initial form display:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> data = {'name': '', 'url': '', 'comment': 'Foo'}
>>> f = CommentForm(data)
>>> f.is_valid()
False
# The form does *not* fall back to using the initial values.
>>> f.errors
{'url': [u'This field is required.'], 'name': [u'This field is required.']}
```

Instead of a constant, you can also pass any callable:

```
>>> import datetime
>>> class DateForm(forms.Form):
...     day = forms.DateField(initial=datetime.date.today)
>>> print(DateForm())
<tr><th>Day:</th><td><input type="text" name="day" value="12/23/2008" /><td></tr>
```

The callable will be evaluated only when the unbound form is displayed, not when it is defined.

widget

Field.**widget**

The `widget` argument lets you specify a `Widget` class to use when rendering this `Field`. See *Widgets* for more information.

help_text

Field.**help_text**

The `help_text` argument lets you specify descriptive text for this `Field`. If you provide `help_text`, it will be displayed next to the `Field` when the `Field` is rendered by one of the convenience `Form` methods (e.g., `as_ul()`).

Here's a full example `Form` that implements `help_text` for two of its fields. We've specified `auto_id=False` to simplify the output:

```
>>> class HelpTextContactForm(forms.Form):
...     subject = forms.CharField(max_length=100, help_text='100 characters max.')
...     message = forms.CharField()
...     sender = forms.EmailField(help_text='A valid email address, please.')
...     cc_myself = forms.BooleanField(required=False)
>>> f = HelpTextContactForm(auto_id=False)
>>> print(f.as_table())
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" /><br /><span class="helpt
<tr><th>Message:</th><td><input type="text" name="message" /></td></tr>
<tr><th>Sender:</th><td><input type="text" name="sender" /><br />A valid email address, please.</td><
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print(f.as_ul()))
<li>Subject: <input type="text" name="subject" maxlength="100" /> <span class="helptext">100 characte
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="text" name="sender" /> A valid email address, please.</li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print(f.as_p())
<p>Subject: <input type="text" name="subject" maxlength="100" /> <span class="helptext">100 characte
```

```
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="text" name="sender" /> A valid email address, please.</p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

error_messages

Field.**error_messages**

The `error_messages` argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override. For example, here is the default error message:

```
>>> generic = forms.CharField()
>>> generic.clean('')
Traceback (most recent call last):
  ...
ValidationError: [u'This field is required.']
```

And here is a custom error message:

```
>>> name = forms.CharField(error_messages={'required': 'Please enter your name'})
>>> name.clean('')
Traceback (most recent call last):
  ...
ValidationError: [u'Please enter your name']
```

In the built-in Field classes section below, each `Field` defines the error message keys it uses.

validators

Field.**validators**

The `validators` argument lets you provide a list of validation functions for this field.

See the *validators documentation* for more information.

localize

Field.**localize**

The `localize` argument enables the localization of form data, input as well as the rendered output.

See the *format localization* documentation for more information.

Built-in `Field` classes

Naturally, the `forms` library comes with a set of `Field` classes that represent common validation needs. This section documents each built-in field.

For each field, we describe the default widget used if you don't specify `widget`. We also specify the value returned when you provide an empty value (see the section on `required` above to understand what that means).

BooleanField

class BooleanField(***kwargs*)

- Default widget: `CheckboxInput`
- Empty value: `False`
- Normalizes to: A Python `True` or `False` value.
- Validates that the value is `True` (e.g. the check box is checked) if the field has `required=True`.
- Error message keys: `required`

Note: Since all `Field` subclasses have `required=True` by default, the validation condition here is important. If you want to include a boolean in your form that can be either `True` or `False` (e.g. a checked or unchecked checkbox), you must remember to pass in `required=False` when creating the `BooleanField`.

CharField

class CharField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates `max_length` or `min_length`, if they are provided. Otherwise, all inputs are valid.
- Error message keys: `required`, `max_length`, `min_length`

Has two optional arguments for validation:

max_length

min_length

If provided, these arguments ensure that the string is at most or at least the given length.

ChoiceField

class ChoiceField(***kwargs*)

- Default widget: `Select`
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value exists in the list of choices.
- Error message keys: `required`, `invalid_choice`

The `invalid_choice` error message may contain `%(value)s`, which will be replaced with the selected choice.

Takes one extra required argument:

choices
An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field. This argument accepts the same formats as the `choices` argument to a model field. See the *model field reference documentation on choices* for more details.

TypedChoiceField

class TypedChoiceField(***kwargs*)
Just like a `ChoiceField`, except `TypedChoiceField` takes two extra arguments, `coerce` and `empty_value`.

- Default widget: `Select`
- Empty value: Whatever you've given as `empty_value`
- Normalizes to: A value of the type provided by the `coerce` argument.
- Validates that the given value exists in the list of choices and can be coerced.
- Error message keys: `required`, `invalid_choice`

Takes extra arguments:

coerce
A function that takes one argument and returns a coerced value. Examples include the built-in `int`, `float`, `bool` and other types. Defaults to an identity function.

empty_value
The value to use to represent "empty." Defaults to the empty string; `None` is another common choice here. Note that this value will not be coerced by the function given in the `coerce` argument, so choose it accordingly.

DateField

class DateField(***kwargs*)

- Default widget: `DateInput`
- Empty value: `None`
- Normalizes to: A Python `datetime.date` object.
- Validates that the given value is either a `datetime.date`, `datetime.datetime` or string formatted in a particular date format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

input_formats
A list of formats used to attempt to convert a string to a valid `datetime.date` object.

If no `input_formats` argument is provided, the default input formats are:

```
'%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', # '2006-10-25', '10/25/2006', '10/25/06'
'%b %d %Y', '%b %d, %Y',            # 'Oct 25 2006', 'Oct 25, 2006'
'%d %b %Y', '%d %b, %Y',            # '25 Oct 2006', '25 Oct, 2006'
'%B %d %Y', '%B %d, %Y',            # 'October 25 2006', 'October 25, 2006'
'%d %B %Y', '%d %B, %Y',            # '25 October 2006', '25 October, 2006'
```

DateTimeField

class DateTimeField(***kwargs*)

- Default widget: `DateTimeInput`
- Empty value: `None`
- Normalizes to: A Python `datetime.datetime` object.
- Validates that the given value is either a `datetime.datetime`, `datetime.date` or string formatted in a particular datetime format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

input_formats
A list of formats used to attempt to convert a string to a valid `datetime.datetime` object.

If no `input_formats` argument is provided, the default input formats are:

```
'%Y-%m-%d %H:%M:%S',     # '2006-10-25 14:30:59'
'%Y-%m-%d %H:%M',        # '2006-10-25 14:30'
'%Y-%m-%d',              # '2006-10-25'
'%m/%d/%Y %H:%M:%S',     # '10/25/2006 14:30:59'
'%m/%d/%Y %H:%M',        # '10/25/2006 14:30'
'%m/%d/%Y',              # '10/25/2006'
'%m/%d/%y %H:%M:%S',     # '10/25/06 14:30:59'
'%m/%d/%y %H:%M',        # '10/25/06 14:30'
'%m/%d/%y',              # '10/25/06'
```

DecimalField

class DecimalField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python `decimal`.
- Validates that the given value is a decimal. Leading and trailing whitespace is ignored.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`, `max_digits`, `max_decimal_places`, `max_whole_digits`

The `max_value` and `min_value` error messages may contain `%(limit_value)s`, which will be substituted by the appropriate limit.

Takes four optional arguments:

max_value

min_value
These control the range of values permitted in the field, and should be given as `decimal.Decimal` values.

max_digits
The maximum number of digits (those before the decimal point plus those after the decimal point, with leading zeros stripped) permitted in the value.

decimal_places
The maximum number of decimal places permitted.

EmailField

class EmailField(***kwargs*)

•Default widget: `TextInput`

•Empty value: `''` (an empty string)

•Normalizes to: A Unicode object.

•Validates that the given value is a valid email address, using a moderately complex regular expression.

•Error message keys: `required`, `invalid`

Has two optional arguments for validation, `max_length` and `min_length`. If provided, these arguments ensure that the string is at most or at least the given length.

FileField

class FileField(***kwargs*)

•Default widget: `ClearableFileInput`

•Empty value: `None`

•Normalizes to: An `UploadedFile` object that wraps the file content and file name into a single object.

•Can validate that non-empty file data has been bound to the form.

•Error message keys: `required`, `invalid`, `missing`, `empty`, `max_length`

Has two optional arguments for validation, `max_length` and `allow_empty_file`. If provided, these ensure that the file name is at most the given length, and that validation will succeed even if the file content is empty.

To learn more about the `UploadedFile` object, see the *file uploads documentation*.

When you use a `FileField` in a form, you must also remember to *bind the file data to the form*.

The `max_length` error refers to the length of the filename. In the error message for that key, `%(max)d` will be replaced with the maximum filename length and `%(length)d` will be replaced with the current filename length.

FilePathField

class FilePathField(***kwargs*)

•Default widget: `Select`

•Empty value: `None`

•Normalizes to: A unicode object

•Validates that the selected choice exists in the list of choices.

•Error message keys: `required`, `invalid_choice`

The field allows choosing from files inside a certain directory. It takes three extra arguments; only `path` is required:

path
The absolute path to the directory whose contents you want listed. This directory must exist.

recursive
If `False` (the default) only the direct contents of `path` will be offered as choices. If `True`, the directory will be descended into recursively and all descendants will be listed as choices.

match
A regular expression pattern; only files with names matching this expression will be allowed as choices.

allow_files
New in version 1.5: *Please see the release notes* Optional. Either `True` or `False`. Default is `True`. Specifies whether files in the specified location should be included. Either this or `allow_folders` must be `True`.

allow_folders
New in version 1.5: *Please see the release notes* Optional. Either `True` or `False`. Default is `False`. Specifies whether folders in the specified location should be included. Either this or `allow_files` must be `True`.

FloatField

class FloatField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python float.
- Validates that the given value is an float. Leading and trailing whitespace is allowed, as in Python's `float()` function.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`

Takes two optional arguments for validation, `max_value` and `min_value`. These control the range of values permitted in the field.

ImageField

class ImageField(***kwargs*)

- Default widget: `ClearableFileInput`
- Empty value: `None`
- Normalizes to: An `UploadedFile` object that wraps the file content and file name into a single object.
- Validates that file data has been bound to the form, and that the file is of an image format understood by PIL.
- Error message keys: `required`, `invalid`, `missing`, `empty`, `invalid_image`

Using an `ImageField` requires that the Python Imaging Library (PIL) is installed and supports the image formats you use. If you encounter a `corrupt image` error when you upload an image, it usually means PIL doesn't understand its format. To fix this, install the appropriate library and reinstall PIL.

When you use an `ImageField` on a form, you must also remember to *bind the file data to the form*.

IntegerField

class IntegerField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python integer or long integer.
- Validates that the given value is an integer. Leading and trailing whitespace is allowed, as in Python's `int()` function.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`

The `max_value` and `min_value` error messages may contain `%(limit_value)s`, which will be substituted by the appropriate limit.

Takes two optional arguments for validation:

max_value

min_value

These control the range of values permitted in the field.

IPAddressField

class IPAddressField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value is a valid IPv4 address, using a regular expression.
- Error message keys: `required`, `invalid`

GenericIPAddressField

New in version 1.4: *Please see the release notes*

class GenericIPAddressField(***kwargs*)

A field containing either an IPv4 or an IPv6 address.

- Default widget: `TextInput`
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object. IPv6 addresses are normalized as described below.
- Validates that the given value is a valid IP address.
- Error message keys: `required`, `invalid`

The IPv6 address normalization follows **RFC 4291** section 2.2, including using the IPv4 format suggested in paragraph 3 of that section, like `::ffff:192.0.2.0`. For example, `2001:0::0:01` would be normalized to `2001::1`, and `::ffff:0a0a:0a0a` to `::ffff:10.10.10.10`. All characters are converted to lowercase.

Takes two optional arguments:

protocol
Limits valid inputs to the specified protocol. Accepted values are `both` (default), `IPv4` or `IPv6`. Matching is case insensitive.

unpack_ipv4
Unpacks IPv4 mapped addresses like `::ffff::192.0.2.1`. If this option is enabled that address would be unpacked to `192.0.2.1`. Default is disabled. Can only be used when `protocol` is set to `'both'`.

MultipleChoiceField

class MultipleChoiceField(***kwargs*)

- Default widget: `SelectMultiple`
- Empty value: `[]` (an empty list)
- Normalizes to: A list of Unicode objects.
- Validates that every value in the given list of values exists in the list of choices.
- Error message keys: `required`, `invalid_choice`, `invalid_list`

The `invalid_choice` error message may contain `%(value)s`, which will be replaced with the selected choice.

Takes one extra required argument, `choices`, as for `ChoiceField`.

TypedMultipleChoiceField

New in version 1.3: *Please see the release notes*

class TypedMultipleChoiceField(***kwargs*)
Just like a `MultipleChoiceField`, except `TypedMultipleChoiceField` takes two extra arguments, `coerce` and `empty_value`.

- Default widget: `SelectMultiple`
- Empty value: Whatever you've given as `empty_value`
- Normalizes to: A list of values of the type provided by the `coerce` argument.
- Validates that the given values exists in the list of choices and can be coerced.
- Error message keys: `required`, `invalid_choice`

The `invalid_choice` error message may contain `%(value)s`, which will be replaced with the selected choice.

Takes two extra arguments, `coerce` and `empty_value`, as for `TypedChoiceField`.

NullBooleanField

class NullBooleanField(***kwargs*)

- Default widget: `NullBooleanSelect`
- Empty value: `None`
- Normalizes to: A Python `True`, `False` or `None` value.

•Validates nothing (i.e., it never raises a `ValidationError`).

RegexField

class RegexField(***kwargs*)

•Default widget: `TextInput`

•Empty value: `''` (an empty string)

•Normalizes to: A Unicode object.

•Validates that the given value matches against a certain regular expression.

•Error message keys: `required`, `invalid`

Takes one required argument:

regex
A regular expression specified either as a string or a compiled regular expression object.

Also takes `max_length` and `min_length`, which work just as they do for `CharField`.

The optional argument `error_message` is also accepted for backwards compatibility. The preferred way to provide an error message is to use the `error_messages` argument, passing a dictionary with `'invalid'` as a key and the error message as the value.

SlugField

class SlugField(***kwargs*)

•Default widget: `TextInput`

•Empty value: `''` (an empty string)

•Normalizes to: A Unicode object.

•Validates that the given value contains only letters, numbers, underscores, and hyphens.

•Error messages: `required`, `invalid`

This field is intended for use in representing a model `SlugField` in forms.

TimeField

class TimeField(***kwargs*)

•Default widget: `TextInput`

•Empty value: `None`

•Normalizes to: A Python `datetime.time` object.

•Validates that the given value is either a `datetime.time` or string formatted in a particular time format.

•Error message keys: `required`, `invalid`

Takes one optional argument:

input_formats
A list of formats used to attempt to convert a string to a valid `datetime.time` object.

If no `input_formats` argument is provided, the default input formats are:

```
'%H:%M:%S',     # '14:30:59'
'%H:%M',        # '14:30'
```

URLField

class URLField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value is a valid URL.
- Error message keys: `required`, `invalid`

Takes the following optional arguments:

max_length

min_length

These are the same as `CharField.max_length` and `CharField.min_length`.

Slightly complex built-in Field classes

ComboField

class ComboField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `''` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value against each of the fields specified as an argument to the `ComboField`.
- Error message keys: `required`, `invalid`

Takes one extra required argument:

fields
The list of fields that should be used to validate the field's value (in the order in which they are provided).

```
>>> f = ComboField(fields=[CharField(max_length=20), EmailField()])
>>> f.clean('test@example.com')
u'test@example.com'
>>> f.clean('longemailaddress@example.com')
Traceback (most recent call last):
...
ValidationError: [u'Ensure this value has at most 20 characters (it has 28).']
```

MultiValueField

class MultiValueField(***kwargs*)

- Default widget: `TextInput`
- Empty value: `''` (an empty string)
- Normalizes to: the type returned by the `compress` method of the subclass.
- Validates that the given value against each of the fields specified as an argument to the `MultiValueField`.
- Error message keys: `required`, `invalid`

This abstract field (must be subclassed) aggregates the logic of multiple fields. Subclasses should not have to implement clean(). Instead, they must implement compress(), which takes a list of valid values and returns a "compressed" version of those values – a single value. For example, `SplitDateTimeField` is a subclass which combines a time field and a date field into a datetime object.

Takes one extra required argument:

`fields`
A list of fields which are cleaned into a single field. Each value in `clean` is cleaned by the corresponding field in `fields` – the first value is cleaned by the first field, the second value is cleaned by the second field, etc. Once all fields are cleaned, the list of clean values is "compressed" into a single value.

`SplitDateTimeField`

class `SplitDateTimeField`(***kwargs*)

- Default widget: `SplitDateTimeWidget`
- Empty value: `None`
- Normalizes to: A Python `datetime.datetime` object.
- Validates that the given value is a `datetime.datetime` or string formatted in a particular datetime format.
- Error message keys: `required`, `invalid`, `invalid_date`, `invalid_time`

Takes two optional arguments:

`input_date_formats`
A list of formats used to attempt to convert a string to a valid `datetime.date` object.

If no `input_date_formats` argument is provided, the default input formats for `DateField` are used.

`input_time_formats`
A list of formats used to attempt to convert a string to a valid `datetime.time` object.

If no `input_time_formats` argument is provided, the default input formats for `TimeField` are used.

Fields which handle relationships

Two fields are available for representing relationships between models: `ModelChoiceField` and `ModelMultipleChoiceField`. Both of these fields require a single `queryset` parameter that is used to create the choices for the field. Upon form validation, these fields will place either one model object (in the case of `ModelChoiceField`) or multiple model objects (in the case of `ModelMultipleChoiceField`) into the `cleaned_data` dictionary of the form.

ModelChoiceField

class ModelChoiceField(***kwargs*)

- Default widget: `Select`
- Empty value: `None`
- Normalizes to: A model instance.
- Validates that the given id exists in the queryset.
- Error message keys: `required`, `invalid_choice`

Allows the selection of a single model object, suitable for representing a foreign key. Note that the default widget for `ModelChoiceField` becomes impractical when the number of entries increases. You should avoid using it for more than 100 items.

A single argument is required:

queryset
A `QuerySet` of model objects from which the choices for the field will be derived, and which will be used to validate the user's selection.

`ModelChoiceField` also takes one optional argument:

empty_label
By default the `<select>` widget used by `ModelChoiceField` will have an empty choice at the top of the list. You can change the text of this label (which is `"---------"` by default) with the `empty_label` attribute, or you can disable the empty label entirely by setting `empty_label` to `None`:

```
# A custom empty label
field1 = forms.ModelChoiceField(queryset=..., empty_label="(Nothing)")

# No empty label
field2 = forms.ModelChoiceField(queryset=..., empty_label=None)
```

Note that if a `ModelChoiceField` is required and has a default initial value, no empty choice is created (regardless of the value of `empty_label`).

The `__unicode__` method of the model will be called to generate string representations of the objects for use in the field's choices; to provide customized representations, subclass `ModelChoiceField` and override `label_from_instance`. This method will receive a model object, and should return a string suitable for representing it. For example:

```
class MyModelChoiceField(ModelChoiceField):
    def label_from_instance(self, obj):
        return "My Object #%i" % obj.id
```

ModelMultipleChoiceField

class ModelMultipleChoiceField(***kwargs*)

- Default widget: `SelectMultiple`
- Empty value: `[]` (an empty list)
- Normalizes to: A list of model instances.
- Validates that every id in the given list of values exists in the queryset.
- Error message keys: `required`, `list`, `invalid_choice`, `invalid_pk_value`

Allows the selection of one or more model objects, suitable for representing a many-to-many relation. As with `ModelChoiceField`, you can use `label_from_instance` to customize the object representations, and `queryset` is a required parameter:

queryset
A `QuerySet` of model objects from which the choices for the field will be derived, and which will be used to validate the user's selection.

Creating custom fields

If the built-in `Field` classes don't meet your needs, you can easily create custom `Field` classes. To do this, just create a subclass of `django.forms.Field`. Its only requirements are that it implement a `clean()` method and that its `__init__()` method accept the core arguments mentioned above (`required`, `label`, `initial`, `widget`, `help_text`).

6.10.3 Widgets

A widget is Django's representation of a HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget.

Specifying widgets

Whenever you specify a field on a form, Django will use a default widget that is appropriate to the type of data that is to be displayed. To find which widget is used on which field, see the documentation about *Built-in Field classes*.

However, if you want to use a different widget for a field, you can just use the `widget` argument on the field definition. For example:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

This would specify a form with a comment that uses a larger `Textarea` widget, rather than the default `TextInput` widget.

Setting arguments for widgets

Many widgets have optional extra arguments; they can be set when defining the widget on the field. In the following example, the `years` attribute is set for a `SelectDateWidget`:

```
from django.forms.fields import DateField, ChoiceField, MultipleChoiceField
from django.forms.widgets import RadioSelect, CheckboxSelectMultiple
from django.forms.extras.widgets import SelectDateWidget

BIRTH_YEAR_CHOICES = ('1980', '1981', '1982')
FAVORITE_COLORS_CHOICES = (('blue', 'Blue'),
                           ('green', 'Green'),
                           ('black', 'Black'))

class SimpleForm(forms.Form):
    birth_year = DateField(widget=SelectDateWidget(years=BIRTH_YEAR_CHOICES))
```

```
    favorite_colors = forms.MultipleChoiceField(required=False,
        widget=CheckboxSelectMultiple, choices=FAVORITE_COLORS_CHOICES)
```

See the *Built-in widgets* for more information about which widgets are available and which arguments they accept.

Widgets inheriting from the Select widget

Widgets inheriting from the `Select` widget deal with choices. They present the user with a list of options to choose from. The different widgets present this choice differently; the `Select` widget itself uses a `<select>` HTML list representation, while `RadioSelect` uses radio buttons.

`Select` widgets are used by default on `ChoiceField` fields. The choices displayed on the widget are inherited from the `ChoiceField` and changing `ChoiceField.choices` will update `Select.choices`. For example:

```
>>> from django import forms
>>> CHOICES = (('1', 'First',), ('2', 'Second',))
>>> choice_field = forms.ChoiceField(widget=forms.RadioSelect, choices=CHOICES)
>>> choice_field.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices
[('1', 'First'), ('2', 'Second')]
>>> choice_field.widget.choices = ()
>>> choice_field.choices = (('1', 'First and only',),)
>>> choice_field.widget.choices
[('1', 'First and only')]
```

Widgets which offer a `choices` attribute can however be used with fields which are not based on choice – such as a `CharField` – but it is recommended to use a `ChoiceField`-based field when the choices are inherent to the model and not just the representational widget.

Customizing widget instances

When Django renders a widget as HTML, it only renders the bare minimum HTML - Django doesn't add a class definition, or any other widget-specific attributes. This means that all `TextInput` widgets will appear the same on your Web page.

If you want to make one widget look different to another, you need to specify additional attributes for each widget. When you specify a widget, you can provide a list of attributes that will be added to the rendered HTML for the widget.

For example, take the following simple form:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField()
```

This form will include three default `TextInput` widgets, with default rendering – no CSS class, no extra attributes. This means that the input boxes provided for each widget will be rendered exactly the same:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" /></td></tr>
<tr><th>Url:</th><td><input type="text" name="url"/></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

On a real Web page, you probably don't want every widget to look the same. You might want a larger input element for the comment, and you might want the 'name' widget to have some special CSS class. To do this, you use the `Widget.attrs` argument when creating the widget:

For example:

```
class CommentForm(forms.Form):
    name = forms.CharField(
                widget=forms.TextInput(attrs={'class':'special'}))
    url = forms.URLField()
    comment = forms.CharField(
                widget=forms.TextInput(attrs={'size':'40'}))
```

Django will then include the extra attributes in the rendered output:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" class="special"/></td></tr>
<tr><th>Url:</th><td><input type="text" name="url"/></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size="40"/></td></tr>
```

Built-in widgets

Django provides a representation of all the basic HTML widgets, plus some commonly used groups of widgets:

Widget

class Widget

This abstract class cannot be rendered, but provides the basic attribute `attrs`.

attrs

A dictionary containing HTML attributes to be set on the rendered widget.

```
>>> name = forms.TextInput(attrs={'size': 10, 'title': 'Your name',})
>>> name.render('name', 'A name')
u'<input title="Your name" type="text" name="name" value="A name" size="10" />'
```

TextInput

class TextInput

Text input: `<input type='text' ...>`

PasswordInput

class PasswordInput

Password input: `<input type='password' ...>`

Takes one optional argument:

render_value

Determines whether the widget will have a value filled in when the form is re-displayed after a validation error (default is `False`). Changed in version 1.3: The default value for `render_value` was changed from `True` to `False`

HiddenInput

class HiddenInput
Hidden input: `<input type='hidden' ...>`

MultipleHiddenInput

class MultipleHiddenInput
Multiple `<input type='hidden' ...>` widgets.

A widget that handles multiple hidden widgets for fields that have a list of values.

choices
This attribute is optional when the field does not have a `choices` attribute. If it does, it will override anything you set here when the attribute is updated on the `Field`.

FileInput

class FileInput
File upload input: `<input type='file' ...>`

ClearableFileInput

class ClearableFileInput
New in version 1.3: *Please see the release notes* File upload input: `<input type='file' ...>`, with an additional checkbox input to clear the field's value, if the field is not required and has initial data.

DateInput

class DateInput
Date input as a simple text box: `<input type='text' ...>`

Takes one optional argument:

format
The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in `DATE_INPUT_FORMATS` and respects *Format localization*.

DateTimeInput

class DateTimeInput
Date/time input as a simple text box: `<input type='text' ...>`

Takes one optional argument:

format
The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in `DATETIME_INPUT_FORMATS` and respects *Format localization*.

TimeInput

class TimeInput

Time input as a simple text box: `<input type='text' ...>`

Takes one optional argument:

format

The format in which this field's initial value will be displayed.

If no `format` argument is provided, the default format is the first format found in `TIME_INPUT_FORMATS` and respects *Format localization*.

Textarea

class Textarea

Text area: `<textarea>...</textarea>`

CheckboxInput

class CheckboxInput

Checkbox: `<input type='checkbox' ...>`

Takes one optional argument:

check_test

A callable that takes the value of the CheckBoxInput and returns `True` if the checkbox should be checked for that value.

Select

class Select

Select widget: `<select><option ...>...</select>`

choices

This attribute is optional when the field does not have a `choices` attribute. If it does, it will override anything you set here when the attribute is updated on the `Field`.

NullBooleanSelect

class NullBooleanSelect

Select widget with options 'Unknown', 'Yes' and 'No'

SelectMultiple

class SelectMultiple

Similar to `Select`, but allows multiple selection: `<select multiple='multiple'>...</select>`

RadioSelect

class RadioSelect

Similar to `Select`, but rendered as a list of radio buttons within `<li>` tags:

```
<ul>
  <li><input type='radio' ...></li>
  ...
</ul>
```

New in version 1.4: *Please see the release notes* For more granular control over the generated markup, you can loop over the radio buttons in the template. Assuming a form `myform` with a field `beatles` that uses a `RadioSelect` as its widget:

```
{% for radio in myform.beatles %}
<div class="myradio">
    {{ radio }}
</div>
{% endfor %}
```

This would generate the following HTML:

```
<div class="myradio">
    <label><input type="radio" name="beatles" value="john" /> John</label>
</div>
<div class="myradio">
    <label><input type="radio" name="beatles" value="paul" /> Paul</label>
</div>
<div class="myradio">
    <label><input type="radio" name="beatles" value="george" /> George</label>
</div>
<div class="myradio">
    <label><input type="radio" name="beatles" value="ringo" /> Ringo</label>
</div>
```

That included the `<label>` tags. To get more granular, you can use each radio button's `tag` and `choice_label` attributes. For example, this template...

```
{% for radio in myform.beatles %}
    <label>
        {{ radio.choice_label }}
        <span class="radio">{{ radio.tag }}</span>
    </label>
{% endfor %}
```

...will result in the following HTML:

```
<label>
    John
    <span class="radio"><input type="radio" name="beatles" value="john" /></span>
</label>
<label>
    Paul
    <span class="radio"><input type="radio" name="beatles" value="paul" /></span>
</label>
<label>
    George
    <span class="radio"><input type="radio" name="beatles" value="george" /></span>
</label>
```

```
<label>
    Ringo
    <span class="radio"><input type="radio" name="beatles" value="ringo" /></span>
</label>
```

If you decide not to loop over the radio buttons – e.g., if your template simply includes `{{ myform.beatles }}` – they'll be output in a `<ul>` with `<li>` tags, as above.

CheckboxSelectMultiple

class CheckboxSelectMultiple

Similar to `SelectMultiple`, but rendered as a list of check buttons:

```
<ul>
  <li><input type='checkbox' ...></li>
  ...
</ul>
```

MultiWidget

class MultiWidget

Wrapper around multiple other widgets. You'll probably want to use this class with `MultiValueField`.

Its `render()` method is different than other widgets', because it has to figure out how to split a single value for display in multiple widgets.

Subclasses may implement `format_output`, which takes the list of rendered widgets and returns a string of HTML that formats them any way you'd like.

The `value` argument used when rendering can be one of two things:

- A `list`.
- A single value (e.g., a string) that is the "compressed" representation of a `list` of values.

In the second case – i.e., if the value is *not* a list – `render()` will first decompress the value into a `list` before rendering it. It does so by calling the `decompress()` method, which `MultiWidget`'s subclasses must implement. This method takes a single "compressed" value and returns a `list`. An example of this is how `SplitDateTimeWidget` turns a `datetime` value into a list with date and time split into two seperate values:

```
class SplitDateTimeWidget(MultiWidget):

    # ...

    def decompress(self, value):
        if value:
            return [value.date(), value.time().replace(microsecond=0)]
        return [None, None]
```

When `render()` executes its HTML rendering, each value in the list is rendered with the corresponding widget – the first value is rendered in the first widget, the second value is rendered in the second widget, etc.

`MultiWidget` has one required argument:

widgets

An iterable containing the widgets needed.

SplitDateTimeWidget

class SplitDateTimeWidget

Wrapper (using `MultiWidget`) around two widgets: `DateInput` for the date, and `TimeInput` for the time.

`SplitDateTimeWidget` has two optional attributes:

date_format

Similar to `DateInput.format`

time_format

Similar to `TimeInput.format`

SplitHiddenDateTimeWidget

class SplitHiddenDateTimeWidget

Similar to `SplitDateTimeWidget`, but uses `HiddenInput` for both date and time.

SelectDateWidget

class SelectDateWidget

Wrapper around three `Select` widgets: one each for month, day, and year. Note that this widget lives in a separate file from the standard widgets.

Takes one optional argument:

years

An optional list/tuple of years to use in the "year" select box. The default is a list containing the current year and the next 9 years.

6.10.4 Form and field validation

Form validation happens when the data is cleaned. If you want to customize this process, there are various places you can change, each one serving a different purpose. Three types of cleaning methods are run during form processing. These are normally executed when you call the `is_valid()` method on a form. There are other things that can trigger cleaning and validation (accessing the `errors` attribute or calling `full_clean()` directly), but normally they won't be needed.

In general, any cleaning method can raise `ValidationError` if there is a problem with the data it is processing, passing the relevant error message to the `ValidationError` constructor. If no `ValidationError` is raised, the method should return the cleaned (normalized) data as a Python object.

If you detect multiple errors during a cleaning method and wish to signal all of them to the form submitter, it is possible to pass a list of errors to the `ValidationError` constructor.

Most validation can be done using validators - simple helpers that can be reused easily. Validators are simple functions (or callables) that take a single argument and raise `ValidationError` on invalid input. Validators are run after the field's `to_python` and `validate` methods have been called.

Validation of a Form is split into several steps, which can be customized or overridden:

- The `to_python()` method on a Field is the first step in every validation. It coerces the value to correct datatype and raises `ValidationError` if that is not possible. This method accepts the raw value from the widget and returns the converted value. For example, a FloatField will turn the data into a Python `float` or raise a `ValidationError`.

- The `validate()` method on a Field handles field-specific validation that is not suitable for a validator, It takes a value that has been coerced to correct datatype and raises `ValidationError` on any error. This method does not return anything and shouldn't alter the value. You should override it to handle validation logic that you can't or don't want to put in a validator.
- The `run_validators()` method on a Field runs all of the field's validators and aggregates all the errors into a single `ValidationError`. You shouldn't need to override this method.
- The `clean()` method on a Field subclass. This is responsible for running `to_python`, `validate` and `run_validators` in the correct order and propagating their errors. If, at any time, any of the methods raise `ValidationError`, the validation stops and that error is raised. This method returns the clean data, which is then inserted into the `cleaned_data` dictionary of the form.
- The `clean_<fieldname>()` method in a form subclass – where `<fieldname>` is replaced with the name of the form field attribute. This method does any cleaning that is specific to that particular attribute, unrelated to the type of field that it is. This method is not passed any parameters. You will need to look up the value of the field in `self.cleaned_data` and remember that it will be a Python object at this point, not the original string submitted in the form (it will be in `cleaned_data` because the general field `clean()` method, above, has already cleaned the data once).

  For example, if you wanted to validate that the contents of a `CharField` called `serialnumber` was unique, `clean_serialnumber()` would be the right place to do this. You don't need a specific field (it's just a `CharField`), but you want a formfield-specific piece of validation and, possibly, cleaning/normalizing the data.

  Just like the general field `clean()` method, above, this method should return the cleaned data, regardless of whether it changed anything or not.
- The Form subclass's `clean()` method. This method can perform any validation that requires access to multiple fields from the form at once. This is where you might put in things to check that if field `A` is supplied, field `B` must contain a valid email address and the like. The data that this method returns is the final `cleaned_data` attribute for the form, so don't forget to return the full list of cleaned data if you override this method (by default, `Form.clean()` just returns `self.cleaned_data`).

  Note that any errors raised by your `Form.clean()` override will not be associated with any field in particular. They go into a special "field" (called `__all__`), which you can access via the `non_field_errors()` method if you need to. If you want to attach errors to a specific field in the form, you will need to access the `_errors` attribute on the form, which is described later.

  Also note that there are special considerations when overriding the `clean()` method of a `ModelForm` subclass. (see the *ModelForm documentation* for more information)

These methods are run in the order given above, one field at a time. That is, for each field in the form (in the order they are declared in the form definition), the `Field.clean()` method (or its override) is run, then `clean_<fieldname>()`. Finally, once those two methods are run for every field, the `Form.clean()` method, or its override, is executed.

Examples of each of these methods are provided below.

As mentioned, any of these methods can raise a `ValidationError`. For any field, if the `Field.clean()` method raises a `ValidationError`, any field-specific cleaning method is not called. However, the cleaning methods for all remaining fields are still executed.

The `clean()` method for the `Form` class or subclass is always run. If that method raises a `ValidationError`, `cleaned_data` will be an empty dictionary.

The previous paragraph means that if you are overriding `Form.clean()`, you should iterate through `self.cleaned_data.items()`, possibly considering the `_errors` dictionary attribute on the form as well. In this way, you will already know which fields have passed their individual validation requirements.

Form subclasses and modifying field errors

Sometimes, in a form's `clean()` method, you will want to add an error message to a particular field in the form. This won't always be appropriate and the more typical situation is to raise a `ValidationError` from `Form.clean()`, which is turned into a form-wide error that is available through the `Form.non_field_errors()` method.

When you really do need to attach the error to a particular field, you should store (or amend) a key in the `Form._errors` attribute. This attribute is an instance of a `django.forms.util.ErrorDict` class. Essentially, though, it's just a dictionary. There is a key in the dictionary for each field in the form that has an error. Each value in the dictionary is a `django.forms.util.ErrorList` instance, which is a list that knows how to display itself in different ways. So you can treat `_errors` as a dictionary mapping field names to lists.

If you want to add a new error to a particular field, you should check whether the key already exists in `self._errors` or not. If not, create a new entry for the given key, holding an empty `ErrorList` instance. In either case, you can then append your error message to the list for the field name in question and it will be displayed when the form is displayed.

There is an example of modifying `self._errors` in the following section.

What's in a name?

You may be wondering why is this attribute called `_errors` and not `errors`. Normal Python practice is to prefix a name with an underscore if it's not for external usage. In this case, you are subclassing the `Form` class, so you are essentially writing new internals. In effect, you are given permission to access some of the internals of `Form`.

Of course, any code outside your form should never access `_errors` directly. The data is available to external code through the `errors` property, which populates `_errors` before returning it).

Another reason is purely historical: the attribute has been called `_errors` since the early days of the forms module and changing it now (particularly since `errors` is used for the read-only property name) would be inconvenient for a number of reasons. You can use whichever explanation makes you feel more comfortable. The result is the same.

Using validation in practice

The previous sections explained how validation works in general for forms. Since it can sometimes be easier to put things into place by seeing each feature in use, here are a series of small examples that use each of the previous features.

Using validators

Django's form (and model) fields support use of simple utility functions and classes known as validators. These can be passed to a field's constructor, via the field's `validators` argument, or defined on the Field class itself with the `default_validators` attribute.

Simple validators can be used to validate values inside the field, let's have a look at Django's `EmailField`:

```
class EmailField(CharField):
    default_error_messages = {
        'invalid': _('Enter a valid e-mail address.'),
    }
    default_validators = [validators.validate_email]
```

As you can see, `EmailField` is just a `CharField` with customized error message and a validator that validates email addresses. This can also be done on field definition so:

```
email = forms.EmailField()
```

is equivalent to:

```
email = forms.CharField(validators=[validators.validate_email],
        error_messages={'invalid': _('Enter a valid e-mail address.')})
```

Form field default cleaning

Let's firstly create a custom form field that validates its input is a string containing comma-separated email addresses. The full class looks like this:

```
from django import forms
from django.core.validators import validate_email

class MultiEmailField(forms.Field):
    def to_python(self, value):
        "Normalize data to a list of strings."

        # Return an empty list if no input was given.
        if not value:
            return []
        return value.split(',')

    def validate(self, value):
        "Check if value consists only of valid emails."

        # Use the parent's handling of required fields, etc.
        super(MultiEmailField, self).validate(value)

        for email in value:
            validate_email(email)
```

Every form that uses this field will have these methods run before anything else can be done with the field's data. This is cleaning that is specific to this type of field, regardless of how it is subsequently used.

Let's create a simple `ContactForm` to demonstrate how you'd use this field:

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    recipients = MultiEmailField()
    cc_myself = forms.BooleanField(required=False)
```

Simply use `MultiEmailField` like any other form field. When the `is_valid()` method is called on the form, the `MultiEmailField.clean()` method will be run as part of the cleaning process and it will, in turn, call the custom `to_python()` and `validate()` methods.

Cleaning a specific field attribute

Continuing on from the previous example, suppose that in our `ContactForm`, we want to make sure that the `recipients` field always contains the address `"fred@example.com"`. This is validation that is specific to our form, so we don't want to put it into the general `MultiEmailField` class. Instead, we write a cleaning method that operates on the `recipients` field, like so:

```
class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean_recipients(self):
        data = self.cleaned_data['recipients']
        if "fred@example.com" not in data:
            raise forms.ValidationError("You have forgotten about Fred!")

        # Always return the cleaned data, whether you have changed it or
        # not.
        return data
```

Cleaning and validating fields that depend on each other

Suppose we add another requirement to our contact form: if the `cc_myself` field is `True`, the `subject` must contain the word `"help"`. We are performing validation on more than one field at a time, so the form's `clean()` method is a good spot to do this. Notice that we are talking about the `clean()` method on the form here, whereas earlier we were writing a `clean()` method on a field. It's important to keep the field and form difference clear when working out where to validate things. Fields are single data points, forms are a collection of fields.

By the time the form's `clean()` method is called, all the individual field clean methods will have been run (the previous two sections), so `self.cleaned_data` will be populated with any data that has survived so far. So you also need to remember to allow for the fact that the fields you are wanting to validate might not have survived the initial individual field checks.

There are two ways to report any errors from this step. Probably the most common method is to display the error at the top of the form. To create such an error, you can raise a `ValidationError` from the `clean()` method. For example:

```
class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = super(ContactForm, self).clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject:
            # Only do something if both fields are valid so far.
            if "help" not in subject:
                raise forms.ValidationError("Did not send for 'help' in "
                        "the subject despite CC'ing yourself.")

        # Always return the full collection of cleaned data.
        return cleaned_data
```

In this code, if the validation error is raised, the form will display an error message at the top of the form (normally) describing the problem.

Note that the call to `super(ContactForm, self).clean()` in the example code ensures that any validation logic in parent classes is maintained.

The second approach might involve assigning the error message to one of the fields. In this case, let's assign an error message to both the "subject" and "cc_myself" rows in the form display. Be careful when doing this in practice, since it can lead to confusing form output. We're showing what is possible here and leaving it up to you and your designers

to work out what works effectively in your particular situation. Our new code (replacing the previous sample) looks like this:

```
class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = super(ContactForm, self).clean()
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject and "help" not in subject:
            # We know these are not in self._errors now (see discussion
            # below).
            msg = u"Must put 'help' in subject when cc'ing yourself."
            self._errors["cc_myself"] = self.error_class([msg])
            self._errors["subject"] = self.error_class([msg])

            # These fields are no longer valid. Remove them from the
            # cleaned data.
            del cleaned_data["cc_myself"]
            del cleaned_data["subject"]

        # Always return the full collection of cleaned data.
        return cleaned_data
```

As you can see, this approach requires a bit more effort, not withstanding the extra design effort to create a sensible form display. The details are worth noting, however. Firstly, earlier we mentioned that you might need to check if the field name keys already exist in the `_errors` dictionary. In this case, since we know the fields exist in `self.cleaned_data`, they must have been valid when cleaned as individual fields, so there will be no corresponding entries in `_errors`.

Secondly, once we have decided that the combined data in the two fields we are considering aren't valid, we must remember to remove them from the `cleaned_data`.

In fact, Django will currently completely wipe out the `cleaned_data` dictionary if there are any errors in the form. However, this behavior may change in the future, so it's not a bad idea to clean up after yourself in the first place.

6.11 Middleware

This document explains all middleware components that come with Django. For information on how to use them and how to write your own middleware, see the *middleware usage guide*.

6.11.1 Available middleware

Cache middleware

class `UpdateCacheMiddleware`

class `FetchFromCacheMiddleware`

Enable the site-wide cache. If these are enabled, each Django-powered page will be cached for as long as the `CACHE_MIDDLEWARE_SECONDS` setting defines. See the *cache documentation*.

"Common" middleware

class CommonMiddleware

Adds a few conveniences for perfectionists:

- Forbids access to user agents in the `DISALLOWED_USER_AGENTS` setting, which should be a list of strings.
- Performs URL rewriting based on the `APPEND_SLASH` and `PREPEND_WWW` settings.

  If `APPEND_SLASH` is `True` and the initial URL doesn't end with a slash, and it is not found in the URLconf, then a new URL is formed by appending a slash at the end. If this new URL is found in the URLconf, then Django redirects the request to this new URL. Otherwise, the initial URL is processed as usual.

  For example, `foo.com/bar` will be redirected to `foo.com/bar/` if you don't have a valid URL pattern for `foo.com/bar` but *do* have a valid pattern for `foo.com/bar/`.

  If `PREPEND_WWW` is `True`, URLs that lack a leading "www." will be redirected to the same URL with a leading "www."

  Both of these options are meant to normalize URLs. The philosophy is that each URL should exist in one, and only one, place. Technically a URL `foo.com/bar` is distinct from `foo.com/bar/` – a search-engine indexer would treat them as separate URLs – so it's best practice to normalize URLs.
- Sends broken link notification emails to `MANAGERS` if `SEND_BROKEN_LINK_EMAILS` is set to `True`.
- Handles ETags based on the `USE_ETAGS` setting. If `USE_ETAGS` is set to `True`, Django will calculate an ETag for each request by MD5-hashing the page content, and it'll take care of sending `Not Modified` responses, if appropriate.

View metadata middleware

class XViewMiddleware

Sends custom `X-View` HTTP headers to HEAD requests that come from IP addresses defined in the `INTERNAL_IPS` setting. This is used by Django's *automatic documentation system*. Depends on `AuthenticationMiddleware`.

GZip middleware

class GZipMiddleware

Compresses content for browsers that understand GZip compression (all modern browsers).

It is suggested to place this first in the middleware list, so that the compression of the response content is the last thing that happens.

It will NOT compress content if any of the following are true:

- The content body is less than 200 bytes long.
- The response has already set the `Content-Encoding` header.
- The request (the browser) hasn't sent an `Accept-Encoding` header containing `gzip`.
- The request is from Internet Explorer and the `Content-Type` header contains `javascript` or starts with anything other than `text/`. We do this to avoid a bug in early versions of IE that caused decompression not to be performed on certain content types.

You can apply GZip compression to individual views using the `gzip_page()` decorator.

Conditional GET middleware

class ConditionalGetMiddleware

Handles conditional GET operations. If the response has a `ETag` or `Last-Modified` header, and the request has `If-None-Match` or `If-Modified-Since`, the response is replaced by an `HttpNotModified`.

Also sets the `Date` and `Content-Length` response-headers.

Reverse proxy middleware

class SetRemoteAddrFromForwardedFor

This middleware was removed in Django 1.1. See *the release notes* for details.

Locale middleware

class LocaleMiddleware

Enables language selection based on data from the request. It customizes content for each user. See the *internationalization documentation*.

Message middleware

class MessageMiddleware

Enables cookie- and session-based message support. See the *messages documentation*.

Session middleware

class SessionMiddleware

Enables session support. See the *session documentation*.

Authentication middleware

class AuthenticationMiddleware

Adds the `user` attribute, representing the currently-logged-in user, to every incoming `HttpRequest` object. See *Authentication in Web requests*.

CSRF protection middleware

class CsrfViewMiddleware

Adds protection against Cross Site Request Forgeries by adding hidden form fields to POST forms and checking requests for the correct value. See the *Cross Site Request Forgery protection documentation*.

Transaction middleware

class `TransactionMiddleware`

Binds commit and rollback to the request/response phase. If a view function runs successfully, a commit is done. If it fails with an exception, a rollback is done.

The order of this middleware in the stack is important: middleware modules running outside of it run with commit-on-save - the default Django behavior. Middleware modules running inside it (coming later in the stack) will be under the same transaction control as the view functions.

See the *transaction management documentation*.

X-Frame-Options middleware

class `XFrameOptionsMiddleware`

New in version 1.4: `XFrameOptionsMiddleware` was added. Simple *clickjacking protection via the X-Frame-Options header*.

6.12 Models

Model API reference. For introductory material, see *Models*.

6.12.1 Model field reference

This document contains all the gory details about all the field options and field types Django's got to offer.

See Also:

If the built-in fields don't do the trick, you can try `django.contrib.localflavor`, which contains assorted pieces of code that are useful for particular countries or cultures. Also, you can easily *write your own custom model fields*.

Note: Technically, these models are defined in `django.db.models.fields`, but for convenience they're imported into `django.db.models`; the standard convention is to use `from django.db import models` and refer to fields as `models.<Foo>Field`.

Field options

The following arguments are available to all field types. All are optional.

`null`

`Field.`**`null`**

If `True`, Django will store empty values as `NULL` in the database. Default is `False`.

Note that empty string values will always get stored as empty strings, not as `NULL`. Only use `null=True` for non-string fields such as integers, booleans and dates. For both types of fields, you will also need to set `blank=True` if you wish to permit empty values in forms, as the `null` parameter only affects database storage (see `blank`).

Avoid using `null` on string-based fields such as `CharField` and `TextField` unless you have an excellent reason. If a string-based field has `null=True`, that means it has two possible values for "no data": `NULL`, and the empty string. In most cases, it's redundant to have two possible values for "no data;" Django convention is to use the empty string, not `NULL`.

Note: When using the Oracle database backend, the value `NULL` will be stored to denote the empty string regardless of this attribute.

If you want to accept `null` values with `BooleanField`, use `NullBooleanField` instead.

blank

`Field.`**`blank`**

If `True`, the field is allowed to be blank. Default is `False`.

Note that this is different than `null`. `null` is purely database-related, whereas `blank` is validation-related. If a field has `blank=True`, validation on Django's admin site will allow entry of an empty value. If a field has `blank=False`, the field will be required.

choices

`Field.`**`choices`**

An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field.

If this is given, Django's admin will use a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

The first element in each tuple is the actual value to be stored. The second element is the human-readable name for the option.

The choices list can be defined either as part of your model class:

```
class Foo(models.Model):
    YEAR_IN_SCHOOL_CHOICES = (
        ('FR', 'Freshman'),
        ('SO', 'Sophomore'),
        ('JR', 'Junior'),
        ('SR', 'Senior'),
        ('GR', 'Graduate'),
    )
    year_in_school = models.CharField(max_length=2, choices=YEAR_IN_SCHOOL_CHOICES)
```

or outside your model class altogether:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
class Foo(models.Model):
    year_in_school = models.CharField(max_length=2, choices=YEAR_IN_SCHOOL_CHOICES)
```

You can also collect your available choices into named groups that can be used for organizational purposes:

```
MEDIA_CHOICES = (
    ('Audio', (
            ('vinyl', 'Vinyl'),
            ('cd', 'CD'),
        )
    ),
    ('Video', (
            ('vhs', 'VHS Tape'),
            ('dvd', 'DVD'),
        )
    ),
    ('unknown', 'Unknown'),
)
```

The first element in each tuple is the name to apply to the group. The second element is an iterable of 2-tuples, with each 2-tuple containing a value and a human-readable name for an option. Grouped options may be combined with ungrouped options within a single list (such as the *unknown* option in this example).

For each model field that has `choices` set, Django will add a method to retrieve the human-readable name for the field's current value. See `get_FOO_display()` in the database API documentation.

Finally, note that choices can be any iterable object – not necessarily a list or tuple. This lets you construct choices dynamically. But if you find yourself hacking `choices` to be dynamic, you're probably better off using a proper database table with a `ForeignKey`. `choices` is meant for static data that doesn't change much, if ever.

db_column

`Field.`**`db_column`**

The name of the database column to use for this field. If this isn't given, Django will use the field's name.

If your database column name is an SQL reserved word, or contains characters that aren't allowed in Python variable names – notably, the hyphen – that's OK. Django quotes column and table names behind the scenes.

db_index

`Field.`**`db_index`**

If `True`, djadmin:*django-admin.py sqlindexes* <*sqlindexes*> will output a `CREATE INDEX` statement for this field.

db_tablespace

`Field.`**`db_tablespace`**

The name of the *database tablespace* to use for this field's index, if this field is indexed. The default is the project's `DEFAULT_INDEX_TABLESPACE` setting, if set, or the `db_tablespace` of the model, if any. If the backend doesn't support tablespaces for indexes, this option is ignored.

default

`Field.`**`default`**

The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

editable

`Field.`**`editable`**

If `False`, the field will not be editable in the admin or via forms automatically generated from the model class. Default is `True`.

error_messages

`Field.`**`error_messages`**

The `error_messages` argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override.

Error message keys include `null`, `blank`, `invalid`, `invalid_choice`, and `unique`. Additional error message keys are specified for each field in the Field types section below.

help_text

`Field.`**`help_text`**

Extra "help" text to be displayed under the field on the object's admin form. It's useful for documentation even if your object doesn't have an admin form.

Note that this value is *not* HTML-escaped when it's displayed in the admin interface. This lets you include HTML in `help_text` if you so desire. For example:

```
help_text="Please use the following format: <em>YYYY-MM-DD</em>."
```

Alternatively you can use plain text and `django.utils.html.escape()` to escape any HTML special characters.

primary_key

`Field.`**`primary_key`**

If `True`, this field is the primary key for the model.

If you don't specify `primary_key=True` for any fields in your model, Django will automatically add an `IntegerField` to hold the primary key, so you don't need to set `primary_key=True` on any of your fields unless you want to override the default primary-key behavior. For more, see *Automatic primary key fields*.

`primary_key=True` implies `null=False` and `unique=True`. Only one primary key is allowed on an object.

unique

Field.**unique**

If `True`, this field must be unique throughout the table.

This is enforced at the database level and at the Django admin-form level. If you try to save a model with a duplicate value in a `unique` field, a `django.db.IntegrityError` will be raised by the model's `save()` method.

This option is valid on all field types except `ManyToManyField` and `FileField`.

unique_for_date

Field.**unique_for_date**

Set this to the name of a `DateField` or `DateTimeField` to require that this field be unique for the value of the date field.

For example, if you have a field `title` that has `unique_for_date="pub_date"`, then Django wouldn't allow the entry of two records with the same `title` and `pub_date`.

This is enforced at the Django admin-form level but not at the database level.

unique_for_month

Field.**unique_for_month**

Like `unique_for_date`, but requires the field to be unique with respect to the month.

unique_for_year

Field.**unique_for_year**

Like `unique_for_date` and `unique_for_month`.

verbose_name

Field.**verbose_name**

A human-readable name for the field. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces. See *Verbose field names*.

validators

Field.**validators**

A list of validators to run for this field. See the *validators documentation* for more information.

Field types

AutoField

class AutoField(***options*)

An `IntegerField` that automatically increments according to available IDs. You usually won't need to use this directly; a primary key field will automatically be added to your model if you don't specify otherwise. See *Automatic primary key fields*.

BigIntegerField

class BigIntegerField(*[**options]*)

A 64 bit integer, much like an `IntegerField` except that it is guaranteed to fit numbers from -9223372036854775808 to 9223372036854775807. The admin represents this as an `<input type="text">` (a single-line input).

BooleanField

class BooleanField(***options*)

A true/false field.

The admin represents this as a checkbox.

If you need to accept `null` values then use `NullBooleanField` instead.

CharField

class CharField(*max_length=None[, **options]*)

A string field, for small- to large-sized strings.

For large amounts of text, use `TextField`.

The admin represents this as an `<input type="text">` (a single-line input).

`CharField` has one extra required argument:

`CharField`.**max_length**
: The maximum length (in characters) of the field. The max_length is enforced at the database level and in Django's validation.

Note: If you are writing an application that must be portable to multiple database backends, you should be aware that there are restrictions on `max_length` for some backends. Refer to the *database backend notes* for details.

MySQL users

If you are using this field with MySQLdb 1.2.2 and the `utf8_bin` collation (which is *not* the default), there are some issues to be aware of. Refer to the *MySQL database notes* for details.

CommaSeparatedIntegerField

class CommaSeparatedIntegerField(*max_length=None[, **options]*)

A field of integers separated by commas. As in `CharField`, the `max_length` argument is required and the note about database portability mentioned there should be heeded.

DateField

class DateField([*auto_now=False*, *auto_now_add=False*, ***options*])

A date, represented in Python by a `datetime.date` instance. Has a few extra, optional arguments:

`DateField.`**auto_now**
Automatically set the field to now every time the object is saved. Useful for "last-modified" timestamps. Note that the current date is *always* used; it's not just a default value that you can override.

`DateField.`**auto_now_add**
Automatically set the field to now when the object is first created. Useful for creation of timestamps. Note that the current date is *always* used; it's not just a default value that you can override.

The admin represents this as an `<input type="text">` with a JavaScript calendar, and a shortcut for "Today". Includes an additional `invalid_date` error message key.

Note: As currently implemented, setting `auto_now` or `auto_now_add` to `True` will cause the field to have `editable=False` and `blank=True` set.

DateTimeField

class DateTimeField([*auto_now=False*, *auto_now_add=False*, ***options*])

A date and time, represented in Python by a `datetime.datetime` instance. Takes the same extra arguments as `DateField`.

The admin represents this as two `<input type="text">` fields, with JavaScript shortcuts.

DecimalField

class DecimalField(*max_digits=None*, *decimal_places=None*[, ***options*])

A fixed-precision decimal number, represented in Python by a `Decimal` instance. Has two **required** arguments:

`DecimalField.`**max_digits**
The maximum number of digits allowed in the number. Note that this number must be greater than or equal to `decimal_places`, if it exists.

`DecimalField.`**decimal_places**
The number of decimal places to store with the number.

For example, to store numbers up to 999 with a resolution of 2 decimal places, you'd use:

```
models.DecimalField(..., max_digits=5, decimal_places=2)
```

And to store numbers up to approximately one billion with a resolution of 10 decimal places:

```
models.DecimalField(..., max_digits=19, decimal_places=10)
```

The admin represents this as an `<input type="text">` (a single-line input).

Note: For more information about the differences between the `FloatField` and `DecimalField` classes, please see *FloatField vs. DecimalField*.

EmailField

class EmailField(*[max_length=75, **options]*)

A `CharField` that checks that the value is a valid email address.

Incompliance to RFCs

The default 75 character `max_length` is not capable of storing all possible RFC3696/5321-compliant email addresses. In order to store all possible valid email addresses, a `max_length` of 254 is required. The default `max_length` of 75 exists for historical reasons. The default has not been changed in order to maintain backwards compatibility with existing uses of `EmailField`.

FileField

class FileField(*upload_to=None[, max_length=100, **options]*)

A file-upload field.

Note: The `primary_key` and `unique` arguments are not supported, and will raise a `TypeError` if used.

Has one **required** argument:

`FileField.`**upload_to**

A local filesystem path that will be appended to your `MEDIA_ROOT` setting to determine the value of the `url` attribute.

This path may contain `strftime()` formatting, which will be replaced by the date/time of the file upload (so that uploaded files don't fill up the given directory).

This may also be a callable, such as a function, which will be called to obtain the upload path, including the filename. This callable must be able to accept two arguments, and return a Unix-style path (with forward slashes) to be passed along to the storage system. The two arguments that will be passed are:

Argument	Description
`instance`	An instance of the model where the `FileField` is defined. More specifically, this is the particular instance where the current file is being attached. In most cases, this object will not have been saved to the database yet, so if it uses the default `AutoField`, *it might not yet have a value for its primary key field*.
`filename`	The filename that was originally given to the file. This may or may not be taken into account when determining the final destination path.

Also has one optional argument:

`FileField.`**storage**

Optional. A storage object, which handles the storage and retrieval of your files. See *Managing files* for details on how to provide this object.

The admin represents this field as an `<input type="file">` (a file-upload widget).

Using a `FileField` or an `ImageField` (see below) in a model takes a few steps:

1. In your settings file, you'll need to define `MEDIA_ROOT` as the full path to a directory where you'd like Django to store uploaded files. (For performance, these files are not stored in the database.) Define `MEDIA_URL` as the base public URL of that directory. Make sure that this directory is writable by the Web server's user account.

2. Add the `FileField` or `ImageField` to your model, making sure to define the `upload_to` option to tell Django to which subdirectory of `MEDIA_ROOT` it should upload files.
3. All that will be stored in your database is a path to the file (relative to `MEDIA_ROOT`). You'll most likely want to use the convenience `url` function provided by Django. For example, if your `ImageField` is called `mug_shot`, you can get the absolute path to your image in a template with `{{ object.mug_shot.url }}`.

For example, say your `MEDIA_ROOT` is set to `'/home/media'`, and `upload_to` is set to `'photos/%Y/%m/%d'`. The `'%Y/%m/%d'` part of `upload_to` is `strftime()` formatting; `'%Y'` is the four-digit year, `'%m'` is the two-digit month and `'%d'` is the two-digit day. If you upload a file on Jan. 15, 2007, it will be saved in the directory `/home/media/photos/2007/01/15`.

If you wanted to retrieve the uploaded file's on-disk filename, or the file's size, you could use the `name` and `size` attributes respectively; for more information on the available attributes and methods, see the `File` class reference and the *Managing files* topic guide.

Note: The file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until after the model has been saved.

The uploaded file's relative URL can be obtained using the `url` attribute. Internally, this calls the `url()` method of the underlying `Storage` class. Note that whenever you deal with uploaded files, you should pay close attention to where you're uploading them and what type of files they are, to avoid security holes. *Validate all uploaded files* so that you're sure the files are what you think they are. For example, if you blindly let somebody upload files, without validation, to a directory that's within your Web server's document root, then somebody could upload a CGI or PHP script and execute that script by visiting its URL on your site. Don't allow that.

Also note that even an uploaded HTML file, since it can be executed by the browser (though not by the server), can pose security threats that are equivalent to XSS or CSRF attacks.

By default, `FileField` instances are created as `varchar(100)` columns in your database. As with other fields, you can change the maximum length using the `max_length` argument.

FileField and FieldFile When you access a `FileField` on a model, you are given an instance of `FieldFile` as a proxy for accessing the underlying file. This class has several methods that can be used to interact with file data:

`FieldFile.`**`open`**`(`*`mode='rb'`*`)`

Behaves like the standard Python `open()` method and opens the file associated with this instance in the mode specified by `mode`.

`FieldFile.`**`close`**`()`

Behaves like the standard Python `file.close()` method and closes the file associated with this instance.

`FieldFile.`**`save`**`(`*`name`*`, `*`content`*`, `*`save=True`*`)`

This method takes a filename and file contents and passes them to the storage class for the field, then associates the stored file with the model field. If you want to manually associate file data with `FileField` instances on your model, the `save()` method is used to persist that file data.

Takes two required arguments: `name` which is the name of the file, and `content` which is an object containing the file's contents. The optional `save` argument controls whether or not the instance is saved after the file has been altered. Defaults to `True`.

Note that the `content` argument should be an instance of `django.core.files.File`, not Python's built-in file object. You can construct a `File` from an existing Python file object like this:

```
from django.core.files import File
# Open an existing file using Python's built-in open()
f = open('/tmp/hello.world')
myfile = File(f)
```

Or you can construct one from a Python string like this:

```
from django.core.files.base import ContentFile
myfile = ContentFile("hello world")
```

For more information, see *Managing files*.

`FieldFile.`**`delete`**`(`*`save=True`*`)`

Deletes the file associated with this instance and clears all attributes on the field. Note: This method will close the file if it happens to be open when `delete()` is called.

The optional `save` argument controls whether or not the instance is saved after the file has been deleted. Defaults to `True`.

FilePathField

class `FilePathField`(*path=None*[, *match=None*, *recursive=False*, *max_length=100*, ***options*])

A `CharField` whose choices are limited to the filenames in a certain directory on the filesystem. Has three special arguments, of which the first is **required**:

`FilePathField.`**`path`**
: Required. The absolute filesystem path to a directory from which this `FilePathField` should get its choices. Example: `"/home/images"`.

`FilePathField.`**`match`**
: Optional. A regular expression, as a string, that `FilePathField` will use to filter filenames. Note that the regex will be applied to the base filename, not the full path. Example: `"foo.*\.txt$"`, which will match a file called `foo23.txt` but not `bar.txt` or `foo23.png`.

`FilePathField.`**`recursive`**
: Optional. Either `True` or `False`. Default is `False`. Specifies whether all subdirectories of `path` should be included

`FilePathField.`**`allow_files`**
: New in version 1.5: *Please see the release notes* Optional. Either `True` or `False`. Default is `True`. Specifies whether files in the specified location should be included. Either this or `allow_folders` must be `True`.

`FilePathField.`**`allow_folders`**
: New in version 1.5: *Please see the release notes* Optional. Either `True` or `False`. Default is `False`. Specifies whether folders in the specified location should be included. Either this or `allow_files` must be `True`.

Of course, these arguments can be used together.

The one potential gotcha is that `match` applies to the base filename, not the full path. So, this example:

```
FilePathField(path="/home/images", match="foo.*", recursive=True)
```

...will match `/home/images/foo.png` but not `/home/images/foo/bar.png` because the `match` applies to the base filename (`foo.png` and `bar.png`).

By default, `FilePathField` instances are created as `varchar(100)` columns in your database. As with other fields, you can change the maximum length using the `max_length` argument.

FloatField

class FloatField([**options*])

A floating-point number represented in Python by a `float` instance.

The admin represents this as an `<input type="text">` (a single-line input).

FloatField vs. DecimalField

The `FloatField` class is sometimes mixed up with the `DecimalField` class. Although they both represent real numbers, they represent those numbers differently. `FloatField` uses Python's `float` type internally, while `DecimalField` uses Python's `Decimal` type. For information on the difference between the two, see Python's documentation for the `decimal` module.

ImageField

class ImageField(*upload_to=None*[, *height_field=None*, *width_field=None*, *max_length=100*, ***options*])

Inherits all attributes and methods from `FileField`, but also validates that the uploaded object is a valid image.

In addition to the special attributes that are available for `FileField`, an `ImageField` also has `height` and `width` attributes.

To facilitate querying on those attributes, `ImageField` has two extra optional arguments:

`ImageField.`**height_field**
: Name of a model field which will be auto-populated with the height of the image each time the model instance is saved.

`ImageField.`**width_field**
: Name of a model field which will be auto-populated with the width of the image each time the model instance is saved.

Requires the Python Imaging Library.

By default, `ImageField` instances are created as `varchar(100)` columns in your database. As with other fields, you can change the maximum length using the `max_length` argument.

IntegerField

class IntegerField([**options*])

An integer. The admin represents this as an `<input type="text">` (a single-line input).

IPAddressField

class IPAddressField([**options*])

An IP address, in string format (e.g. "192.0.2.30"). The admin represents this as an `<input type="text">` (a single-line input).

GenericIPAddressField

class GenericIPAddressField([*protocol=both*, *unpack_ipv4=False*, ***options*])

New in version 1.4: *Please see the release notes* An IPv4 or IPv6 address, in string format (e.g. `192.0.2.30` or `2a02:42fe::4`). The admin represents this as an `<input type="text">` (a single-line input).

The IPv6 address normalization follows **RFC 4291** section 2.2, including using the IPv4 format suggested in paragraph 3 of that section, like `::ffff:192.0.2.0`. For example, `2001:0::0:01` would be normalized to `2001::1`, and `::ffff:0a0a:0a0a` to `::ffff:10.10.10.10`. All characters are converted to lowercase.

GenericIPAddressField.**protocol**
: Limits valid inputs to the specified protocol. Accepted values are `'both'` (default), `'IPv4'` or `'IPv6'`. Matching is case insensitive.

GenericIPAddressField.**unpack_ipv4**
: Unpacks IPv4 mapped addresses like `::ffff::192.0.2.1`. If this option is enabled that address would be unpacked to `192.0.2.1`. Default is disabled. Can only be used when `protocol` is set to `'both'`.

NullBooleanField

class NullBooleanField([***options*])

Like a `BooleanField`, but allows `NULL` as one of the options. Use this instead of a `BooleanField` with `null=True`. The admin represents this as a `<select>` box with "Unknown", "Yes" and "No" choices.

PositiveIntegerField

class PositiveIntegerField([***options*])

Like an `IntegerField`, but must be either positive or zero (*0*). The value *0* is accepted for backward compatibility reasons.

PositiveSmallIntegerField

class PositiveSmallIntegerField([***options*])

Like a `PositiveIntegerField`, but only allows values under a certain (database-dependent) point.

SlugField

class SlugField([*max_length=50*, ***options*])

Slug is a newspaper term. A slug is a short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs.

Like a CharField, you can specify `max_length` (read the note about database portability and `max_length` in that section, too). If `max_length` is not specified, Django will use a default length of 50.

Implies setting `Field.db_index` to `True`.

It is often useful to automatically prepopulate a SlugField based on the value of some other value. You can do this automatically in the admin using `prepopulated_fields`.

SmallIntegerField

class SmallIntegerField([**options*])

Like an `IntegerField`, but only allows values under a certain (database-dependent) point.

TextField

class TextField([**options*])

A large text field. The admin represents this as a `<textarea>` (a multi-line input).

MySQL users

If you are using this field with MySQLdb 1.2.1p2 and the `utf8_bin` collation (which is *not* the default), there are some issues to be aware of. Refer to the *MySQL database notes* for details.

TimeField

class TimeField([*auto_now=False*, *auto_now_add=False*, **options*])

A time, represented in Python by a `datetime.time` instance. Accepts the same auto-population options as `DateField`.

The admin represents this as an `<input type="text">` with some JavaScript shortcuts.

URLField

class URLField([*max_length=200*, **options*])

A `CharField` for a URL.

The admin represents this as an `<input type="text">` (a single-line input).

Like all `CharField` subclasses, `URLField` takes the optional `max_length`, a default of 200 is used.

Relationship fields

Django also defines a set of fields that represent relations.

ForeignKey

class ForeignKey(*othermodel*[, **options*])

A many-to-one relationship. Requires a positional argument: the class to which the model is related. To create a recursive relationship – an object that has a many-to-one relationship with itself – use `models.ForeignKey('self')`. If you need to create a relationship on a model that has not yet been defined, you can use the name of the model, rather than the model object itself:

```
class Car(models.Model):
    manufacturer = models.ForeignKey('Manufacturer')
    # ...

class Manufacturer(models.Model):
    # ...
```

To refer to models defined in another application, you can explicitly specify a model with the full application label. For example, if the `Manufacturer` model above is defined in another application called `production`, you'd need to use:

```
class Car(models.Model):
    manufacturer = models.ForeignKey('production.Manufacturer')
```

This sort of reference can be useful when resolving circular import dependencies between two applications.

Database Representation Behind the scenes, Django appends `"_id"` to the field name to create its database column name. In the above example, the database table for the `Car` model will have a `manufacturer_id` column. (You can change this explicitly by specifying `db_column`) However, your code should never have to deal with the database column name, unless you write custom SQL. You'll always deal with the field names of your model object.

Arguments `ForeignKey` accepts an extra set of arguments – all optional – that define the details of how the relation works.

`ForeignKey.`**`limit_choices_to`**

A dictionary of lookup arguments and values (see *Making queries*) that limit the available admin choices for this object. Use this with functions from the Python `datetime` module to limit choices of objects by date. For example:

```
limit_choices_to = {'pub_date__lte': datetime.now}
```

only allows the choice of related objects with a `pub_date` before the current date/time to be chosen.

Instead of a dictionary this can also be a `Q` object for more *complex queries*. However, if `limit_choices_to` is a `Q` object then it will only have an effect on the choices available in the admin when the field is not listed in `raw_id_fields` in the `ModelAdmin` for the model.

`ForeignKey.`**`related_name`**

The name to use for the relation from the related object back to this one. See the *related objects documentation* for a full explanation and example. Note that you must set this value when defining relations on *abstract models*; and when you do so *some special syntax* is available.

If you'd prefer Django didn't create a backwards relation, set `related_name` to `'+'`. For example, this will ensure that the `User` model won't get a backwards relation to this model:

```
user = models.ForeignKey(User, related_name='+')
```

`ForeignKey.`**`to_field`**

The field on the related object that the relation is to. By default, Django uses the primary key of the related object.

New in version 1.3: *Please see the release notes*

`ForeignKey.`**`on_delete`**

When an object referenced by a `ForeignKey` is deleted, Django by default emulates the behavior of the SQL constraint `ON DELETE CASCADE` and also deletes the object containing the `ForeignKey`. This behavior can be overridden by specifying the `on_delete` argument. For example, if you have a nullable `ForeignKey` and you want it to be set null when the referenced object is deleted:

```
user = models.ForeignKey(User, blank=True, null=True, on_delete=models.SET_NULL)
```

The possible values for `on_delete` are found in `django.db.models`:

- `CASCADE`: Cascade deletes; the default.
- `PROTECT`: Prevent deletion of the referenced object by raising `django.db.models.ProtectedError`, a subclass of `django.db.IntegrityError`.
- `SET_NULL`: Set the `ForeignKey` null; this is only possible if `null` is `True`.
- `SET_DEFAULT`: Set the `ForeignKey` to its default value; a default for the `ForeignKey` must be set.
- `SET()`: Set the `ForeignKey` to the value passed to `SET()`, or if a callable is passed in, the result of calling it. In most cases, passing a callable will be necessary to avoid executing queries at the time your models.py is imported:

```
def get_sentinel_user():
    return User.objects.get_or_create(username='deleted')[0]

class MyModel(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET(get_sentinel_user))
```

- `DO_NOTHING`: Take no action. If your database backend enforces referential integrity, this will cause an `IntegrityError` unless you manually add a SQL `ON DELETE` constraint to the database field (perhaps using *initial sql*).

ManyToManyField

class ManyToManyField(*othermodel*[, ***options*])

A many-to-many relationship. Requires a positional argument: the class to which the model is related. This works exactly the same as it does for `ForeignKey`, including all the options regarding *recursive* and *lazy* relationships.

Database Representation Behind the scenes, Django creates an intermediary join table to represent the many-to-many relationship. By default, this table name is generated using the name of the many-to-many field and the name of the table for the model that contains it. Since some databases don't support table names above a certain length, these table names will be automatically truncated to 64 characters and a uniqueness hash will be used. This means you might see table names like `author_books_9cdf4`; this is perfectly normal. You can manually provide the name of the join table using the `db_table` option.

Arguments `ManyToManyField` accepts an extra set of arguments – all optional – that control how the relationship functions.

`ManyToManyField.`**related_name**
Same as `ForeignKey.related_name`.

`ManyToManyField.`**limit_choices_to**
Same as `ForeignKey.limit_choices_to`.

`limit_choices_to` has no effect when used on a `ManyToManyField` with a custom intermediate table specified using the `through` parameter.

`ManyToManyField.`**symmetrical**
Only used in the definition of ManyToManyFields on self. Consider the following model:

```
class Person(models.Model):
    friends = models.ManyToManyField("self")
```

When Django processes this model, it identifies that it has a `ManyToManyField` on itself, and as a result, it doesn't add a `person_set` attribute to the `Person` class. Instead, the `ManyToManyField` is assumed to be symmetrical – that is, if I am your friend, then you are my friend.

If you do not want symmetry in many-to-many relationships with `self`, set `symmetrical` to `False`. This will force Django to add the descriptor for the reverse relationship, allowing `ManyToManyField` relationships to be non-symmetrical.

`ManyToManyField.`**`through`**

Django will automatically generate a table to manage many-to-many relationships. However, if you want to manually specify the intermediary table, you can use the `through` option to specify the Django model that represents the intermediate table that you want to use.

The most common use for this option is when you want to associate *extra data with a many-to-many relationship*.

`ManyToManyField.`**`db_table`**

The name of the table to create for storing the many-to-many data. If this is not provided, Django will assume a default name based upon the names of: the table for the model defining the relationship and the name of the field itself.

`OneToOneField`

class `OneToOneField`(*othermodel*[, *parent_link=False*, ***options*])

A one-to-one relationship. Conceptually, this is similar to a `ForeignKey` with `unique=True`, but the "reverse" side of the relation will directly return a single object.

This is most useful as the primary key of a model which "extends" another model in some way; *Multi-table inheritance* is implemented by adding an implicit one-to-one relation from the child model to the parent model, for example.

One positional argument is required: the class to which the model will be related. This works exactly the same as it does for `ForeignKey`, including all the options regarding *recursive* and *lazy* relationships. Additionally, `OneToOneField` accepts all of the extra arguments accepted by `ForeignKey`, plus one extra argument:

`OneToOneField.`**`parent_link`**

When `True` and used in a model which inherits from another (concrete) model, indicates that this field should be used as the link back to the parent class, rather than the extra `OneToOneField` which would normally be implicitly created by subclassing.

6.12.2 Related objects reference

class `RelatedManager`

A "related manager" is a manager used in a one-to-many or many-to-many related context. This happens in two cases:

- The "other side" of a `ForeignKey` relation. That is:

```
class Reporter(models.Model):
    ...

class Article(models.Model):
    reporter = models.ForeignKey(Reporter)
```

In the above example, the methods below will be available on the manager `reporter.article_set`.

- Both sides of a `ManyToManyField` relation:

```
class Topping(models.Model):
    ...

class Pizza(models.Model):
    toppings = models.ManyToManyField(Topping)
```

In this example, the methods below will be available both on `topping.pizza_set` and on `pizza.toppings`.

These related managers have some extra methods:

add(*obj1*[, *obj2*, ...])

Adds the specified model objects to the related object set.

Example:

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e) # Associates Entry e with Blog b.
```

create(***kwargs*)

Creates a new object, saves it and puts it in the related object set. Returns the newly created object:

```
>>> b = Blog.objects.get(id=1)
>>> e = b.entry_set.create(
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )

# No need to call e.save() at this point -- it's already been saved.
```

This is equivalent to (but much simpler than):

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry(
...     blog=b,
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )
>>> e.save(force_insert=True)
```

Note that there's no need to specify the keyword argument of the model that defines the relationship. In the above example, we don't pass the parameter `blog` to `create()`. Django figures out that the new `Entry` object's `blog` field should be set to `b`.

remove(*obj1*[, *obj2*, ...])

Removes the specified model objects from the related object set:

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e) # Disassociates Entry e from Blog b.
```

In order to prevent database inconsistency, this method only exists on `ForeignKey` objects where `null=True`. If the related field can't be set to `None` (`NULL`), then an object can't be removed from a relation without being added to another. In the above example, removing `e` from `b.entry_set()` is equivalent to doing `e.blog = None`, and because the `blog` `ForeignKey` doesn't have `null=True`, this is invalid.

clear()
Removes all objects from the related object set:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.clear()
```

Note this doesn't delete the related objects – it just disassociates them.

Just like `remove()`, `clear()` is only available on `ForeignKey`s where `null=True`.

6.12.3 Model `Meta` options

This document explains all the possible *metadata options* that you can give your model in its internal `class Meta`.

Available `Meta` options

abstract

`Options.`**abstract**
If `abstract = True`, this model will be an *abstract base class*.

app_label

`Options.`**app_label**
If a model exists outside of the standard `models.py` (for instance, if the app's models are in submodules of `myapp.models`), the model must define which app it is part of:

```
app_label = 'myapp'
```

db_table

`Options.`**db_table**
The name of the database table to use for the model:

```
db_table = 'music_album'
```

Table names To save you time, Django automatically derives the name of the database table from the name of your model class and the app that contains it. A model's database table name is constructed by joining the model's "app label" – the name you used in `manage.py startapp` – to the model's class name, with an underscore between them.

For example, if you have an app `bookstore` (as created by `manage.py startapp bookstore`), a model defined as `class Book` will have a database table named `bookstore_book`.

To override the database table name, use the `db_table` parameter in `class Meta`.

If your database table name is an SQL reserved word, or contains characters that aren't allowed in Python variable names – notably, the hyphen – that's OK. Django quotes column and table names behind the scenes.

Use lowercase table names for MySQL

It is strongly advised that you use lowercase table names when you override the table name via `db_table`, particularly if you are using the MySQL backend. See the *MySQL notes* for more details.

db_tablespace

Options.**db_tablespace**
The name of the *database tablespace* to use for this model. The default is the project's DEFAULT_TABLESPACE setting, if set. If the backend doesn't support tablespaces, this option is ignored.

get_latest_by

Options.**get_latest_by**
The name of a DateField or DateTimeField in the model. This specifies the default field to use in your model Manager's latest method.

Example:

```
get_latest_by = "order_date"
```

See the docs for latest() for more.

managed

Options.**managed**
Defaults to True, meaning Django will create the appropriate database tables in syncdb and remove them as part of a reset management command. That is, Django *manages* the database tables' lifecycles.

If False, no database table creation or deletion operations will be performed for this model. This is useful if the model represents an existing table or a database view that has been created by some other means. This is the *only* difference when managed=False. All other aspects of model handling are exactly the same as normal. This includes

1. Adding an automatic primary key field to the model if you don't declare it. To avoid confusion for later code readers, it's recommended to specify all the columns from the database table you are modeling when using unmanaged models.
2. If a model with managed=False contains a ManyToManyField that points to another unmanaged model, then the intermediate table for the many-to-many join will also not be created. However, the intermediary table between one managed and one unmanaged model *will* be created.

   If you need to change this default behavior, create the intermediary table as an explicit model (with managed set as needed) and use the ManyToManyField.through attribute to make the relation use your custom model.

For tests involving models with managed=False, it's up to you to ensure the correct tables are created as part of the test setup.

If you're interested in changing the Python-level behavior of a model class, you *could* use managed=False and create a copy of an existing model. However, there's a better approach for that situation: *Proxy models*.

order_with_respect_to

Options.**order_with_respect_to**
Marks this object as "orderable" with respect to the given field. This is almost always used with related objects to allow them to be ordered with respect to a parent object. For example, if an Answer relates to a Question object, and a question has more than one answer, and the order of answers matters, you'd do this:

```
class Answer(models.Model):
    question = models.ForeignKey(Question)
    # ...

    class Meta:
        order_with_respect_to = 'question'
```

When `order_with_respect_to` is set, two additional methods are provided to retrieve and to set the order of the related objects: `get_RELATED_order()` and `set_RELATED_order()`, where `RELATED` is the lowercased model name. For example, assuming that a `Question` object has multiple related `Answer` objects, the list returned contains the primary keys of the related `Answer` objects:

```
>>> question = Question.objects.get(id=1)
>>> question.get_answer_order()
[1, 2, 3]
```

The order of a `Question` object's related `Answer` objects can be set by passing in a list of `Answer` primary keys:

```
>>> question.set_answer_order([3, 1, 2])
```

The related objects also get two methods, `get_next_in_order()` and `get_previous_in_order()`, which can be used to access those objects in their proper order. Assuming the `Answer` objects are ordered by `id`:

```
>>> answer = Answer.objects.get(id=2)
>>> answer.get_next_in_order()
<Answer: 3>
>>> answer.get_previous_in_order()
<Answer: 1>
```

Changing order_with_respect_to

`order_with_respect_to` adds an additional field/database column named `_order`, so be sure to handle that as you would any other change to your models if you add or change `order_with_respect_to` after your initial `syncdb`.

---

ordering

`Options.`**`ordering`**

The default ordering for the object, for use when obtaining lists of objects:

```
ordering = ['-order_date']
```

This is a tuple or list of strings. Each string is a field name with an optional "-" prefix, which indicates descending order. Fields without a leading "-" will be ordered ascending. Use the string "?" to order randomly.

For example, to order by a `pub_date` field ascending, use this:

```
ordering = ['pub_date']
```

To order by `pub_date` descending, use this:

```
ordering = ['-pub_date']
```

To order by `pub_date` descending, then by `author` ascending, use this:

```
ordering = ['-pub_date', 'author']
```

Changed in version 1.4: The Django admin honors all elements in the list/tuple; before 1.4, only the first one was respected.

permissions

Options.**permissions**

Extra permissions to enter into the permissions table when creating this object. Add, delete and change permissions are automatically created for each object that has `admin` set. This example specifies an extra permission, `can_deliver_pizzas`:

```
permissions = (("can_deliver_pizzas", "Can deliver pizzas"),)
```

This is a list or tuple of 2-tuples in the format `(permission_code, human_readable_permission_name)`.

proxy

Options.**proxy**

If `proxy = True`, a model which subclasses another model will be treated as a *proxy model*.

unique_together

Options.**unique_together**

Sets of field names that, taken together, must be unique:

```
unique_together = (("driver", "restaurant"),)
```

This is a tuple of tuples that must be unique when considered together. It's used in the Django admin and is enforced at the database level (i.e., the appropriate `UNIQUE` statements are included in the `CREATE TABLE` statement).

For convenience, unique_together can be a single tuple when dealing with a single set of fields:

```
unique_together = ("driver", "restaurant")
```

A `ManyToManyField` cannot be included in unique_together. (It's not clear what that would even mean!) If you need to validate uniqueness related to a `ManyToManyField`, try using a signal or an explicit `through` model.

verbose_name

Options.**verbose_name**

A human-readable name for the object, singular:

```
verbose_name = "pizza"
```

If this isn't given, Django will use a munged version of the class name: `CamelCase` becomes `camel case`.

verbose_name_plural

Options.**verbose_name_plural**

The plural name for the object:

```
verbose_name_plural = "stories"
```

If this isn't given, Django will use `verbose_name + "s"`.

6.12.4 Model instance reference

This document describes the details of the `Model` API. It builds on the material presented in the *model* and *database query* guides, so you'll probably want to read and understand those documents before reading this one.

Throughout this reference we'll use the *example Weblog models* presented in the *database query guide*.

Creating objects

To create a new instance of a model, just instantiate it like any other Python class:

class Model(***kwargs*)

The keyword arguments are simply the names of the fields you've defined on your model. Note that instantiating a model in no way touches your database; for that, you need to `save()`.

Note: You may be tempted to customize the model by overriding the `__init__` method. If you do so, however, take care not to change the calling signature as any change may prevent the model instance from being saved. Rather than overriding `__init__`, try using one of these approaches:

1. Add a classmethod on the model class:

```
class Book(models.Model):
    title = models.CharField(max_length=100)

    @classmethod
    def create(cls, title):
        book = cls(title=title)
        # do something with the book
        return book

book = Book.create("Pride and Prejudice")
```

2. Add a method on a custom manager (usually preferred):

```
class BookManager(models.Manager):
    def create_book(title):
        book = self.create(title=title)
        # do something with the book
        return book

class Book(models.Model):
    title = models.CharField(max_length=100)

    objects = BookManager()

book = Book.objects.create_book("Pride and Prejudice")
```

Validating objects

There are three steps involved in validating a model:

1. Validate the model fields
2. Validate the model as a whole
3. Validate the field uniqueness

All three steps are performed when you call a model's `full_clean()` method.

When you use a `ModelForm`, the call to `is_valid()` will perform these validation steps for all the fields that are included on the form. See the *ModelForm documentation* for more information. You should only need to call a model's `full_clean()` method if you plan to handle validation errors yourself, or if you have excluded fields from the `ModelForm` that require validation.

`Model.`**`full_clean`**`(`*exclude=None*`)`

This method calls `Model.clean_fields()`, `Model.clean()`, and `Model.validate_unique()`, in that order and raises a `ValidationError` that has a `message_dict` attribute containing errors from all three stages.

The optional `exclude` argument can be used to provide a list of field names that can be excluded from validation and cleaning. `ModelForm` uses this argument to exclude fields that aren't present on your form from being validated since any errors raised could not be corrected by the user.

Note that `full_clean()` will *not* be called automatically when you call your model's `save()` method, nor as a result of `ModelForm` validation. You'll need to call it manually when you want to run one-step model validation for your own manually created models.

Example:

```
try:
    article.full_clean()
except ValidationError as e:
    # Do something based on the errors contained in e.message_dict.
    # Display them to a user, or handle them programatically.
```

The first step `full_clean()` performs is to clean each individual field.

`Model.`**`clean_fields`**`(`*exclude=None*`)`

This method will validate all fields on your model. The optional `exclude` argument lets you provide a list of field names to exclude from validation. It will raise a `ValidationError` if any fields fail validation.

The second step `full_clean()` performs is to call `Model.clean()`. This method should be overridden to perform custom validation on your model.

`Model.`**`clean`**`()`

This method should be used to provide custom model validation, and to modify attributes on your model if desired. For instance, you could use it to automatically provide a value for a field, or to do validation that requires access to more than a single field:

```
def clean(self):
    from django.core.exceptions import ValidationError
    # Don't allow draft entries to have a pub_date.
    if self.status == 'draft' and self.pub_date is not None:
        raise ValidationError('Draft entries may not have a publication date.')
    # Set the pub_date for published items if it hasn't been set already.
    if self.status == 'published' and self.pub_date is None:
        self.pub_date = datetime.datetime.now()
```

Any `ValidationError` exceptions raised by `Model.clean()` will be stored in a special key error dictionary key, `NON_FIELD_ERRORS`, that is used for errors that are tied to the entire model instead of to a specific field:

```
from django.core.exceptions import ValidationError, NON_FIELD_ERRORS
try:
    article.full_clean()
except ValidationError as e:
    non_field_errors = e.message_dict[NON_FIELD_ERRORS]
```

Finally, `full_clean()` will check any unique constraints on your model.

`Model.`**`validate_unique`**`(`*exclude=None*`)`

This method is similar to `clean_fields()`, but validates all uniqueness constraints on your model instead of individual field values. The optional `exclude` argument allows you to provide a list of field names to exclude from validation. It will raise a `ValidationError` if any fields fail validation.

Note that if you provide an `exclude` argument to `validate_unique()`, any `unique_together` constraint involving one of the fields you provided will not be checked.

Saving objects

To save an object back to the database, call `save()`:

`Model.`**`save`**`(`[*force_insert=False*, *force_update=False*, *using=DEFAULT_DB_ALIAS*, *update_fields=None*])

If you want customized saving behavior, you can override this `save()` method. See *Overriding predefined model methods* for more details.

The model save process also has some subtleties; see the sections below.

Auto-incrementing primary keys

If a model has an `AutoField` — an auto-incrementing primary key — then that auto-incremented value will be calculated and saved as an attribute on your object the first time you call `save()`:

```
>>> b2 = Blog(name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b2.id     # Returns None, because b doesn't have an ID yet.
>>> b2.save()
>>> b2.id     # Returns the ID of your new object.
```

There's no way to tell what the value of an ID will be before you call `save()`, because that value is calculated by your database, not by Django.

For convenience, each model has an `AutoField` named `id` by default unless you explicitly specify `primary_key=True` on a field in your model. See the documentation for `AutoField` for more details.

The `pk` property

`Model.`**`pk`**

Regardless of whether you define a primary key field yourself, or let Django supply one for you, each model will have a property called `pk`. It behaves like a normal attribute on the model, but is actually an alias for whichever attribute is the primary key field for the model. You can read and set this value, just as you would for any other attribute, and it will update the correct field in the model.

Explicitly specifying auto-primary-key values If a model has an `AutoField` but you want to define a new object's ID explicitly when saving, just define it explicitly before saving, rather than relying on the auto-assignment of the ID:

```
>>> b3 = Blog(id=3, name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b3.id     # Returns 3.
>>> b3.save()
>>> b3.id     # Returns 3.
```

If you assign auto-primary-key values manually, make sure not to use an already-existing primary-key value! If you create a new object with an explicit primary-key value that already exists in the database, Django will assume you're changing the existing record rather than creating a new one.

Given the above `'Cheddar Talk'` blog example, this example would override the previous record in the database:

```
b4 = Blog(id=3, name='Not Cheddar', tagline='Anything but cheese.')
b4.save()  # Overrides the previous blog with ID=3!
```

See How Django knows to UPDATE vs. INSERT, below, for the reason this happens.

Explicitly specifying auto-primary-key values is mostly useful for bulk-saving objects, when you're confident you won't have primary-key collision.

What happens when you save?

When you save an object, Django performs the following steps:

1. **Emit a pre-save signal.** The *signal* `django.db.models.signals.pre_save` is sent, allowing any functions listening for that signal to take some customized action.
2. **Pre-process the data.** Each field on the object is asked to perform any automated data modification that the field may need to perform.

   Most fields do *no* pre-processing — the field data is kept as-is. Pre-processing is only used on fields that have special behavior. For example, if your model has a `DateField` with `auto_now=True`, the pre-save phase will alter the data in the object to ensure that the date field contains the current date stamp. (Our documentation doesn't yet include a list of all the fields with this "special behavior.")
3. **Prepare the data for the database.** Each field is asked to provide its current value in a data type that can be written to the database.

   Most fields require *no* data preparation. Simple data types, such as integers and strings, are 'ready to write' as a Python object. However, more complex data types often require some modification.

   For example, `DateField` fields use a Python `datetime` object to store data. Databases don't store `datetime` objects, so the field value must be converted into an ISO-compliant date string for insertion into the database.
4. **Insert the data into the database.** The pre-processed, prepared data is then composed into an SQL statement for insertion into the database.
5. **Emit a post-save signal.** The signal `django.db.models.signals.post_save` is sent, allowing any functions listening for that signal to take some customized action.

How Django knows to UPDATE vs. INSERT

You may have noticed Django database objects use the same `save()` method for creating and changing objects. Django abstracts the need to use `INSERT` or `UPDATE` SQL statements. Specifically, when you call `save()`, Django follows this algorithm:

- If the object's primary key attribute is set to a value that evaluates to `True` (i.e., a value other than `None` or the empty string), Django executes a `SELECT` query to determine whether a record with the given primary key already exists.
- If the record with the given primary key does already exist, Django executes an `UPDATE` query.
- If the object's primary key attribute is *not* set, or if it's set but a record doesn't exist, Django executes an `INSERT`.

The one gotcha here is that you should be careful not to specify a primary-key value explicitly when saving new objects, if you cannot guarantee the primary-key value is unused. For more on this nuance, see Explicitly specifying auto-primary-key values above and Forcing an INSERT or UPDATE below.

Forcing an INSERT or UPDATE In some rare circumstances, it's necessary to be able to force the `save()` method to perform an SQL `INSERT` and not fall back to doing an `UPDATE`. Or vice-versa: update, if possible, but not insert a new row. In these cases you can pass the `force_insert=True` or `force_update=True` parameters to the `save()` method. Obviously, passing both parameters is an error: you cannot both insert *and* update at the same time!

It should be very rare that you'll need to use these parameters. Django will almost always do the right thing and trying to override that will lead to errors that are difficult to track down. This feature is for advanced use only.

Using `update_fields` will force an update similarly to `force_update`.

Updating attributes based on existing fields

Sometimes you'll need to perform a simple arithmetic task on a field, such as incrementing or decrementing the current value. The obvious way to achieve this is to do something like:

```
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese')
>>> product.number_sold += 1
>>> product.save()
```

If the old `number_sold` value retrieved from the database was 10, then the value of 11 will be written back to the database.

This sequence has a standard update problem in that it contains a race condition. If another thread of execution has already saved an updated value after the current thread retrieved the old value, the current thread will only save the old value plus one, rather than the new (current) value plus one.

The process can be made robust and slightly faster by expressing the update relative to the original field value, rather than as an explicit assignment of a new value. Django provides *F() expressions* for performing this kind of relative update. Using `F()` expressions, the previous example is expressed as:

```
>>> from django.db.models import F
>>> product = Product.objects.get(name='Venezuelan Beaver Cheese')
>>> product.number_sold = F('number_sold') + 1
>>> product.save()
```

This approach doesn't use the initial value from the database. Instead, it makes the database do the update based on whatever value is current at the time that the `save()` is executed.

Once the object has been saved, you must reload the object in order to access the actual value that was applied to the updated field:

```
>>> product = Products.objects.get(pk=product.pk)
>>> print(product.number_sold)
42
```

For more details, see the documentation on *F() expressions* and their *use in update queries*.

Specifying which fields to save

New in version 1.5: *Please see the release notes* If `save()` is passed a list of field names in keyword argument `update_fields`, only the fields named in that list will be updated. This may be desirable if you want to update just one or a few fields on an object. There will be a slight performance benefit from preventing all of the model fields from being updated in the database. For example:

product.name = 'Name changed again' product.save(update_fields=['name'])

The `update_fields` argument can be any iterable containing strings. An empty `update_fields` iterable will skip the save. A value of None will perform an update on all fields.

Specifying `update_fields` will force an update.

Deleting objects

`Model.`**`delete`**`(`[*using=DEFAULT_DB_ALIAS*]`)`

Issues a SQL `DELETE` for the object. This only deletes the object in the database; the Python instance will still exist and will still have data in its fields.

For more details, including how to delete objects in bulk, see *Deleting objects*.

If you want customized deletion behavior, you can override the `delete()` method. See *Overriding predefined model methods* for more details.

Other model instance methods

A few object methods have special purposes.

`__unicode__`

`Model.`**`__unicode__`**`()`

The `__unicode__()` method is called whenever you call `unicode()` on an object. Django uses `unicode(obj)` (or the related function, `str(obj)`) in a number of places. Most notably, to display an object in the Django admin site and as the value inserted into a template when it displays an object. Thus, you should always return a nice, human-readable representation of the model from the `__unicode__()` method.

For example:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

If you define a `__unicode__()` method on your model and not a `__str__()` method, Django will automatically provide you with a `__str__()` that calls `__unicode__()` and then converts the result correctly to a UTF-8 encoded string object. This is recommended development practice: define only `__unicode__()` and let Django take care of the conversion to string objects when required.

__str__

Model.__str__()

The __str__() method is called whenever you call str() on an object. The main use for this method directly inside Django is when the repr() output of a model is displayed anywhere (for example, in debugging output). Thus, you should return a nice, human-readable string for the object's __str__(). It isn't required to put __str__() methods everywhere if you have sensible __unicode__() methods.

The previous __unicode__() example could be similarly written using __str__() like this:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __str__(self):
        # Note use of django.utils.encoding.smart_str() here because
        # first_name and last_name will be unicode strings.
        return smart_str('%s %s' % (self.first_name, self.last_name))
```

get_absolute_url

Model.**get_absolute_url**()

Define a get_absolute_url() method to tell Django how to calculate the canonical URL for an object. To callers, this method should appear to return a string that can be used to refer to the object over HTTP.

For example:

```
def get_absolute_url(self):
    return "/people/%i/" % self.id
```

(Whilst this code is correct and simple, it may not be the most portable way to write this kind of method. The permalink() decorator, documented below, is usually the best approach and you should read that section before diving into code implementation.)

One place Django uses get_absolute_url() is in the admin app. If an object defines this method, the object-editing page will have a "View on site" link that will jump you directly to the object's public view, as given by get_absolute_url().

Similarly, a couple of other bits of Django, such as the *syndication feed framework*, use get_absolute_url() when it is defined. If it makes sense for your model's instances to each have a unique URL, you should define get_absolute_url().

It's good practice to use get_absolute_url() in templates, instead of hard-coding your objects' URLs. For example, this template code is bad:

```
<!-- BAD template code. Avoid! -->
<a href="/people/{{ object.id }}/">{{ object.name }}</a>
```

This template code is much better:

```
<a href="{{ object.get_absolute_url }}">{{ object.name }}</a>
```

The logic here is that if you change the URL structure of your objects, even for something simple such as correcting a spelling error, you don't want to have to track down every place that the URL might be created. Specify it once, in get_absolute_url() and have all your other code call that one place.

Note: The string you return from `get_absolute_url()` **must** contain only ASCII characters (required by the URI specfication, RFC 2396) and be URL-encoded, if necessary.

Code and templates calling `get_absolute_url()` should be able to use the result directly without any further processing. You may wish to use the `django.utils.encoding.iri_to_uri()` function to help with this if you are using unicode strings containing characters outside the ASCII range at all.

The `permalink` decorator The way we wrote `get_absolute_url()` above is a slightly violation of the DRY principle: the URL for this object is defined both in the URLconf file and in the model.

You can decouple your models from the URLconf using the `permalink` decorator:

`permalink``()`

This decorator takes the name of a URL pattern (either a view name or a URL pattern name) and a list of position or keyword arguments and uses the URLconf patterns to construct the correct, full URL. It returns a string for the correct URL, with all parameters substituted in the correct positions.

The `permalink` decorator is a Python-level equivalent to the `url` template tag and a high-level wrapper for the `django.core.urlresolvers.reverse()` function.

An example should make it clear how to use `permalink()`. Suppose your URLconf contains a line such as:

```
(r'^people/(\d+)/$', 'people.views.details'),
```

...your model could have a `get_absolute_url()` method that looked like this:

```
from django.db import models

@models.permalink
def get_absolute_url(self):
    return ('people.views.details', [str(self.id)])
```

Similarly, if you had a URLconf entry that looked like:

```
(r'/archive/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/$', archive_view)
```

...you could reference this using `permalink()` as follows:

```
@models.permalink
def get_absolute_url(self):
    return ('archive_view', (), {
        'year': self.created.year,
        'month': self.created.strftime('%m'),
        'day': self.created.strftime('%d')})
```

Notice that we specify an empty sequence for the second parameter in this case, because we only want to pass keyword parameters, not positional ones.

In this way, you're associating the model's absolute path with the view that is used to display it, without repeating the view's URL information anywhere. You can still use the `get_absolute_url()` method in templates, as before.

In some cases, such as the use of generic views or the re-use of custom views for multiple models, specifying the view function may confuse the reverse URL matcher (because multiple patterns point to the same view). For that case, Django has *named URL patterns*. Using a named URL pattern, it's possible to give a name to a pattern, and then reference the name rather than the view function. A named URL pattern is defined by replacing the pattern tuple by a call to the `url` function):

```
from django.conf.urls import patterns, url, include

url(r'^people/(\d+)/$', 'blog_views.generic_detail', name='people_view'),
```

...and then using that name to perform the reverse URL resolution instead of the view name:

```
from django.db import models

@models.permalink
def get_absolute_url(self):
    return ('people_view', [str(self.id)])
```

More details on named URL patterns are in the *URL dispatch documentation*.

Extra instance methods

In addition to `save()`, `delete()`, a model object might have some of the following methods:

`Model.`**`get_FOO_display`**`()`

For every field that has `choices` set, the object will have a `get_FOO_display()` method, where `FOO` is the name of the field. This method returns the "human-readable" value of the field.

For example:

```
    from django.db import models

    class Person(models.Model):
        SHIRT_SIZES = (
            (u'S', u'Small'),
            (u'M', u'Medium'),
            (u'L', u'Large'),
        )
        name = models.CharField(max_length=60)
        shirt_size = models.CharField(max_length=2, choices=SHIRT_SIZES)
```

::

```
    >>> p = Person(name="Fred Flintstone", shirt_size="L")
    >>> p.save()
    >>> p.shirt_size
    u'L'
    >>> p.get_shirt_size_display()
    u'Large'
```

`Model.`**`get_next_by_FOO`**`(`*`**kwargs`*`)`

`Model.`**`get_previous_by_FOO`**`(`*`**kwargs`*`)`

For every `DateField` and `DateTimeField` that does not have `null=True`, the object will have `get_next_by_FOO()` and `get_previous_by_FOO()` methods, where `FOO` is the name of the field. This returns the next and previous object with respect to the date field, raising a `DoesNotExist` exception when appropriate.

Both methods accept optional keyword arguments, which should be in the format described in *Field lookups*.

Note that in the case of identical date values, these methods will use the primary key as a tie-breaker. This guarantees that no records are skipped or duplicated. That also means you cannot use those methods on unsaved objects.

6.12.5 QuerySet API reference

This document describes the details of the `QuerySet` API. It builds on the material presented in the *model* and *database query* guides, so you'll probably want to read and understand those documents before reading this one.

Throughout this reference we'll use the *example Weblog models* presented in the *database query guide*.

When QuerySets are evaluated

Internally, a `QuerySet` can be constructed, filtered, sliced, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset.

You can evaluate a `QuerySet` in the following ways:

- **Iteration.** A `QuerySet` is iterable, and it executes its database query the first time you iterate over it. For example, this will print the headline of all entries in the database:

  ```
  for e in Entry.objects.all():
      print(e.headline)
  ```

- **Slicing.** As explained in *Limiting QuerySets*, a `QuerySet` can be sliced, using Python's array-slicing syntax. Slicing an unevaluated `QuerySet` usually returns another unevaluated `QuerySet`, but Django will execute the database query if you use the "step" parameter of slice syntax, and will return a list. Slicing a `QuerySet` that has been evaluated (partially or fully) also returns a list.

- **Pickling/Caching.** See the following section for details of what is involved when pickling QuerySets. The important thing for the purposes of this section is that the results are read from the database.

- **repr().** A `QuerySet` is evaluated when you call `repr()` on it. This is for convenience in the Python interactive interpreter, so you can immediately see your results when using the API interactively.

- **len().** A `QuerySet` is evaluated when you call `len()` on it. This, as you might expect, returns the length of the result list.

  Note: *Don't* use `len()` on `QuerySet`s if all you want to do is determine the number of records in the set. It's much more efficient to handle a count at the database level, using SQL's `SELECT COUNT(*)`, and Django provides a `count()` method for precisely this reason. See `count()` below.

- **list().** Force evaluation of a `QuerySet` by calling `list()` on it. For example:

  ```
  entry_list = list(Entry.objects.all())
  ```

  Be warned, though, that this could have a large memory overhead, because Django will load each element of the list into memory. In contrast, iterating over a `QuerySet` will take advantage of your database to load data and instantiate objects only as you need them.

- **bool().** Testing a `QuerySet` in a boolean context, such as using `bool()`, `or`, `and` or an `if` statement, will cause the query to be executed. If there is at least one result, the `QuerySet` is `True`, otherwise `False`. For example:

  ```
  if Entry.objects.filter(headline="Test"):
     print("There is at least one Entry with the headline Test")
  ```

  Note: *Don't* use this if all you want to do is determine if at least one result exists, and don't need the actual objects. It's more efficient to use `exists()` (see below).

Pickling QuerySets

If you `pickle` a `QuerySet`, this will force all the results to be loaded into memory prior to pickling. Pickling is usually used as a precursor to caching and when the cached queryset is reloaded, you want the results to already be present and ready for use (reading from the database can take some time, defeating the purpose of caching). This means that when you unpickle a `QuerySet`, it contains the results at the moment it was pickled, rather than the results that are currently in the database.

If you only want to pickle the necessary information to recreate the `QuerySet` from the database at a later time, pickle the `query` attribute of the `QuerySet`. You can then recreate the original `QuerySet` (without any results loaded) using some code like this:

```
>>> import pickle
>>> query = pickle.loads(s)      # Assuming 's' is the pickled string.
>>> qs = MyModel.objects.all()
>>> qs.query = query             # Restore the original 'query'.
```

The `query` attribute is an opaque object. It represents the internals of the query construction and is not part of the public API. However, it is safe (and fully supported) to pickle and unpickle the attribute's contents as described here.

You can't share pickles between versions

Pickles of QuerySets are only valid for the version of Django that was used to generate them. If you generate a pickle using Django version N, there is no guarantee that pickle will be readable with Django version N+1. Pickles should not be used as part of a long-term archival strategy.

QuerySet API

Though you usually won't create one manually — you'll go through a `Manager` — here's the formal declaration of a `QuerySet`:

class `QuerySet` ([*model=None*, *query=None*, *using=None*])

Usually when you'll interact with a `QuerySet` you'll use it by *chaining filters*. To make this work, most `QuerySet` methods return new querysets. These methods are covered in detail later in this section.

The `QuerySet` class has two public attributes you can use for introspection:

`ordered`

`True` if the `QuerySet` is ordered — i.e. has an `order_by()` clause or a default ordering on the model. `False` otherwise.

`db`

The database that will be used if this query is executed now.

Note: The `query` parameter to `QuerySet` exists so that specialized query subclasses such as `GeoQuerySet` can reconstruct internal query state. The value of the parameter is an opaque representation of that query state and is not part of a public API. To put it simply: if you need to ask, you don't need to use it.

Methods that return new QuerySets

Django provides a range of `QuerySet` refinement methods that modify either the types of results returned by the `QuerySet` or the way its SQL query is executed.

filter

filter(***kwargs*)

Returns a new QuerySet containing objects that match the given lookup parameters.

The lookup parameters (**kwargs) should be in the format described in Field lookups below. Multiple parameters are joined via AND in the underlying SQL statement.

exclude

exclude(***kwargs*)

Returns a new QuerySet containing objects that do *not* match the given lookup parameters.

The lookup parameters (**kwargs) should be in the format described in Field lookups below. Multiple parameters are joined via AND in the underlying SQL statement, and the whole thing is enclosed in a NOT().

This example excludes all entries whose pub_date is later than 2005-1-3 AND whose headline is "Hello":

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3), headline='Hello')
```

In SQL terms, that evaluates to:

```
SELECT ...
WHERE NOT (pub_date > '2005-1-3' AND headline = 'Hello')
```

This example excludes all entries whose pub_date is later than 2005-1-3 OR whose headline is "Hello":

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3)).exclude(headline='Hello')
```

In SQL terms, that evaluates to:

```
SELECT ...
WHERE NOT pub_date > '2005-1-3'
AND NOT headline = 'Hello'
```

Note the second example is more restrictive.

annotate

annotate(**args*, ***kwargs*)

Annotates each object in the QuerySet with the provided list of aggregate values (averages, sums, etc) that have been computed over the objects that are related to the objects in the QuerySet. Each argument to annotate() is an annotation that will be added to each object in the QuerySet that is returned.

The aggregation functions that are provided by Django are described in Aggregation Functions below.

Annotations specified using keyword arguments will use the keyword as the alias for the annotation. Anonymous arguments will have an alias generated for them based upon the name of the aggregate function and the model field that is being aggregated.

For example, if you were manipulating a list of blogs, you may want to determine how many entries have been made in each blog:

```
>>> q = Blog.objects.annotate(Count('entry'))
# The name of the first blog
>>> q[0].name
'Blogasaurus'
# The number of entries on the first blog
>>> q[0].entry__count
42
```

The Blog model doesn't define an entry__count attribute by itself, but by using a keyword argument to specify the aggregate function, you can control the name of the annotation:

```
>>> q = Blog.objects.annotate(number_of_entries=Count('entry'))
# The number of entries on the first blog, using the name provided
>>> q[0].number_of_entries
42
```

For an in-depth discussion of aggregation, see *the topic guide on Aggregation*.

order_by

order_by(**fields*)

By default, results returned by a `QuerySet` are ordered by the ordering tuple given by the `ordering` option in the model's `Meta`. You can override this on a per-`QuerySet` basis by using the `order_by` method.

Example:

```
Entry.objects.filter(pub_date__year=2005).order_by('-pub_date', 'headline')
```

The result above will be ordered by `pub_date` descending, then by `headline` ascending. The negative sign in front of `"-pub_date"` indicates *descending* order. Ascending order is implied. To order randomly, use `"?"`, like so:

```
Entry.objects.order_by('?')
```

Note: `order_by('?')` queries may be expensive and slow, depending on the database backend you're using.

To order by a field in a different model, use the same syntax as when you are querying across model relations. That is, the name of the field, followed by a double underscore (__), followed by the name of the field in the new model, and so on for as many models as you want to join. For example:

```
Entry.objects.order_by('blog__name', 'headline')
```

If you try to order by a field that is a relation to another model, Django will use the default ordering on the related model (or order by the related model's primary key if there is no `Meta.ordering` specified. For example:

```
Entry.objects.order_by('blog')
```

...is identical to:

```
Entry.objects.order_by('blog__id')
```

...since the `Blog` model has no default ordering specified.

Be cautious when ordering by fields in related models if you are also using `distinct()`. See the note in `distinct()` for an explanation of how related model ordering can change the expected results.

It is permissible to specify a multi-valued field to order the results by (for example, a `ManyToManyField` field). Normally this won't be a sensible thing to do and it's really an advanced usage feature. However, if you know that your queryset's filtering or available data implies that there will only be one ordering piece of data for each of the main items you are selecting, the ordering may well be exactly what you want to do. Use ordering on multi-valued fields with care and make sure the results are what you expect.

There's no way to specify whether ordering should be case sensitive. With respect to case-sensitivity, Django will order results however your database backend normally orders them.

If you don't want any ordering to be applied to a query, not even the default ordering, call `order_by()` with no parameters.

You can tell if a query is ordered or not by checking the `QuerySet.ordered` attribute, which will be `True` if the `QuerySet` has been ordered in any way.

reverse

reverse()

Use the reverse() method to reverse the order in which a queryset's elements are returned. Calling reverse() a second time restores the ordering back to the normal direction.

To retrieve the ''last" five items in a queryset, you could do this:

```
my_queryset.reverse()[:5]
```

Note that this is not quite the same as slicing from the end of a sequence in Python. The above example will return the last item first, then the penultimate item and so on. If we had a Python sequence and looked at seq[-5:], we would see the fifth-last item first. Django doesn't support that mode of access (slicing from the end), because it's not possible to do it efficiently in SQL.

Also, note that reverse() should generally only be called on a QuerySet which has a defined ordering (e.g., when querying against a model which defines a default ordering, or when using order_by()). If no such ordering is defined for a given QuerySet, calling reverse() on it has no real effect (the ordering was undefined prior to calling reverse(), and will remain undefined afterward).

distinct

distinct([**fields*])

Returns a new QuerySet that uses SELECT DISTINCT in its SQL query. This eliminates duplicate rows from the query results.

By default, a QuerySet will not eliminate duplicate rows. In practice, this is rarely a problem, because simple queries such as Blog.objects.all() don't introduce the possibility of duplicate result rows. However, if your query spans multiple tables, it's possible to get duplicate results when a QuerySet is evaluated. That's when you'd use distinct().

Note: Any fields used in an order_by() call are included in the SQL SELECT columns. This can sometimes lead to unexpected results when used in conjunction with distinct(). If you order by fields from a related model, those fields will be added to the selected columns and they may make otherwise duplicate rows appear to be distinct. Since the extra columns don't appear in the returned results (they are only there to support ordering), it sometimes looks like non-distinct results are being returned.

Similarly, if you use a values() query to restrict the columns selected, the columns used in any order_by() (or default model ordering) will still be involved and may affect uniqueness of the results.

The moral here is that if you are using distinct() be careful about ordering by related models. Similarly, when using distinct() and values() together, be careful when ordering by fields not in the values() call.

New in version 1.4: *Please see the release notes* As of Django 1.4, you can pass positional arguments (*fields) in order to specify the names of fields to which the DISTINCT should apply. This translates to a SELECT DISTINCT ON SQL query.

Here's the difference. For a normal distinct() call, the database compares *each* field in each row when determining which rows are distinct. For a distinct() call with specified field names, the database will only compare the specified field names.

Note: This ability to specify field names is only available in PostgreSQL.

Note: When you specify field names, you *must* provide an order_by() in the QuerySet, and the fields in order_by() must start with the fields in distinct(), in the same order.

For example, `SELECT DISTINCT ON (a)` gives you the first row for each value in column `a`. If you don't specify an order, you'll get some arbitrary row.

Examples:

```
>>> Author.objects.distinct()
[...]

>>> Entry.objects.order_by('pub_date').distinct('pub_date')
[...]

>>> Entry.objects.order_by('blog').distinct('blog')
[...]

>>> Entry.objects.order_by('author', 'pub_date').distinct('author', 'pub_date')
[...]

>>> Entry.objects.order_by('blog__name', 'mod_date').distinct('blog__name', 'mod_date')
[...]

>>> Entry.objects.order_by('author', 'pub_date').distinct('author')
[...]
```

values

values(**fields*)

Returns a `ValuesQuerySet` — a `QuerySet` subclass that returns dictionaries when used as an iterable, rather than model-instance objects.

Each of those dictionaries represents an object, with the keys corresponding to the attribute names of model objects.

This example compares the dictionaries of `values()` with the normal model objects:

```
# This list contains a Blog object.
>>> Blog.objects.filter(name__startswith='Beatles')
[<Blog: Beatles Blog>]

# This list contains a dictionary.
>>> Blog.objects.filter(name__startswith='Beatles').values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]
```

The `values()` method takes optional positional arguments, `*fields`, which specify field names to which the `SELECT` should be limited. If you specify the fields, each dictionary will contain only the field keys/values for the fields you specify. If you don't specify the fields, each dictionary will contain a key and value for every field in the database table.

Example:

```
>>> Blog.objects.values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}],
>>> Blog.objects.values('id', 'name')
[{'id': 1, 'name': 'Beatles Blog'}]
```

A few subtleties that are worth mentioning:

- If you have a field called `foo` that is a `ForeignKey`, the default `values()` call will return a dictionary key called `foo_id`, since this is the name of the hidden model attribute that stores the actual value (the `foo` attribute refers to the related model). When you are calling `values()` and passing in field names, you can pass in either `foo` or `foo_id` and you will get back the same thing (the dictionary key will match the field name you passed in).

For example:

```
>>> Entry.objects.values()
[{'blog_id': 1, 'headline': u'First Entry', ...}, ...]

>>> Entry.objects.values('blog')
[{'blog': 1}, ...]

>>> Entry.objects.values('blog_id')
[{'blog_id': 1}, ...]
```

- When using `values()` together with `distinct()`, be aware that ordering can affect the results. See the note in `distinct()` for details.
- If you use a `values()` clause after an `extra()` call, any fields defined by a `select` argument in the `extra()` must be explicitly included in the `values()` call. Any `extra()` call made after a `values()` call will have its extra selected fields ignored.

A `ValuesQuerySet` is useful when you know you're only going to need values from a small number of the available fields and you won't need the functionality of a model instance object. It's more efficient to select only the fields you need to use.

Finally, note a `ValuesQuerySet` is a subclass of `QuerySet`, so it has all methods of `QuerySet`. You can call `filter()` on it, or `order_by()`, or whatever. Yes, that means these two calls are identical:

```
Blog.objects.values().order_by('id')
Blog.objects.order_by('id').values()
```

The people who made Django prefer to put all the SQL-affecting methods first, followed (optionally) by any output-affecting methods (such as `values()`), but it doesn't really matter. This is your chance to really flaunt your individualism. Changed in version 1.3: *Please see the release notes* The `values()` method previously did not return anything for `ManyToManyField` attributes and would raise an error if you tried to pass this type of field to it.

This restriction has been lifted, and you can now also refer to fields on related models with reverse relations through `OneToOneField`, `ForeignKey` and `ManyToManyField` attributes:

```
Blog.objects.values('name', 'entry__headline')
[{'name': 'My blog', 'entry__headline': 'An entry'},
     {'name': 'My blog', 'entry__headline': 'Another entry'}, ...]
```

Warning: Because `ManyToManyField` attributes and reverse relations can have multiple related rows, including these can have a multiplier effect on the size of your result set. This will be especially pronounced if you include multiple such fields in your `values()` query, in which case all possible combinations will be returned.

values_list

`values_list`(**fields*)

This is similar to `values()` except that instead of returning dictionaries, it returns tuples when iterated over. Each tuple contains the value from the respective field passed into the `values_list()` call — so the first item is the first field, etc. For example:

```
>>> Entry.objects.values_list('id', 'headline')
[(1, u'First entry'), ...]
```

If you only pass in a single field, you can also pass in the `flat` parameter. If `True`, this will mean the returned results are single values, rather than one-tuples. An example should make the difference clearer:

```
>>> Entry.objects.values_list('id').order_by('id')
[(1,), (2,), (3,), ...]
```

```
>>> Entry.objects.values_list('id', flat=True).order_by('id')
[1, 2, 3, ...]
```

It is an error to pass in `flat` when there is more than one field.

If you don't pass any values to `values_list()`, it will return all the fields in the model, in the order they were declared.

dates

`dates`(*field*, *kind*, *order='ASC'*)

Returns a `DateQuerySet` — a `QuerySet` that evaluates to a list of `datetime.datetime` objects representing all available dates of a particular kind within the contents of the `QuerySet`.

`field` should be the name of a `DateField` or `DateTimeField` of your model.

`kind` should be either `"year"`, `"month"` or `"day"`. Each `datetime.datetime` object in the result list is "truncated" to the given `type`.

- `"year"` returns a list of all distinct year values for the field.
- `"month"` returns a list of all distinct year/month values for the field.
- `"day"` returns a list of all distinct year/month/day values for the field.

`order`, which defaults to `'ASC'`, should be either `'ASC'` or `'DESC'`. This specifies how to order the results.

Examples:

```
>>> Entry.objects.dates('pub_date', 'year')
[datetime.datetime(2005, 1, 1)]
>>> Entry.objects.dates('pub_date', 'month')
[datetime.datetime(2005, 2, 1), datetime.datetime(2005, 3, 1)]
>>> Entry.objects.dates('pub_date', 'day')
[datetime.datetime(2005, 2, 20), datetime.datetime(2005, 3, 20)]
>>> Entry.objects.dates('pub_date', 'day', order='DESC')
[datetime.datetime(2005, 3, 20), datetime.datetime(2005, 2, 20)]
>>> Entry.objects.filter(headline__contains='Lennon').dates('pub_date', 'day')
[datetime.datetime(2005, 3, 20)]
```

Warning: When *time zone support* is enabled, Django uses UTC in the database connection, which means the aggregation is performed in UTC. This is a known limitation of the current implementation.

none

`none`()

Returns an `EmptyQuerySet` — a `QuerySet` subclass that always evaluates to an empty list. This can be used in cases where you know that you should return an empty result set and your caller is expecting a `QuerySet` object (instead of returning an empty list, for example.)

Examples:

```
>>> Entry.objects.none()
[]
```

all

`all`()

Returns a *copy* of the current `QuerySet` (or `QuerySet` subclass). This can be useful in situations where you might want to pass in either a model manager or a `QuerySet` and do further filtering on the result. After calling `all()` on either object, you'll definitely have a `QuerySet` to work with.

select_related

`select_related``()`

Returns a `QuerySet` that will automatically "follow" foreign-key relationships, selecting that additional related-object data when it executes its query. This is a performance booster which results in (sometimes much) larger queries but means later use of foreign-key relationships won't require database queries.

The following examples illustrate the difference between plain lookups and `select_related()` lookups. Here's standard lookup:

```
# Hits the database.
e = Entry.objects.get(id=5)

# Hits the database again to get the related Blog object.
b = e.blog
```

And here's `select_related` lookup:

```
# Hits the database.
e = Entry.objects.select_related().get(id=5)

# Doesn't hit the database, because e.blog has been prepopulated
# in the previous query.
b = e.blog
```

`select_related()` follows foreign keys as far as possible. If you have the following models:

```
class City(models.Model):
    # ...
    pass

class Person(models.Model):
    # ...
    hometown = models.ForeignKey(City)

class Book(models.Model):
    # ...
    author = models.ForeignKey(Person)
```

...then a call to `Book.objects.select_related().get(id=4)` will cache the related `Person` *and* the related `City`:

```
b = Book.objects.select_related().get(id=4)
p = b.author         # Doesn't hit the database.
c = p.hometown       # Doesn't hit the database.

b = Book.objects.get(id=4) # No select_related() in this example.
p = b.author         # Hits the database.
c = p.hometown       # Hits the database.
```

Note that, by default, `select_related()` does not follow foreign keys that have `null=True`.

Usually, using `select_related()` can vastly improve performance because your app can avoid many database calls. However, in situations with deeply nested sets of relationships `select_related()` can sometimes end up following "too many" relations, and can generate queries so large that they end up being slow.

In these situations, you can use the `depth` argument to `select_related()` to control how many "levels" of relations `select_related()` will actually follow:

```
b = Book.objects.select_related(depth=1).get(id=4)
p = b.author         # Doesn't hit the database.
c = p.hometown       # Requires a database call.
```

Sometimes you only want to access specific models that are related to your root model, not all of the related models. In these cases, you can pass the related field names to `select_related()` and it will only follow those relations. You can even do this for models that are more than one relation away by separating the field names with double underscores, just as for filters. For example, if you have this model:

```
class Room(models.Model):
    # ...
    building = models.ForeignKey(...)

class Group(models.Model):
    # ...
    teacher = models.ForeignKey(...)
    room = models.ForeignKey(Room)
    subject = models.ForeignKey(...)
```

...and you only needed to work with the `room` and `subject` attributes, you could write this:

```
g = Group.objects.select_related('room', 'subject')
```

This is also valid:

```
g = Group.objects.select_related('room__building', 'subject')
```

...and would also pull in the `building` relation.

You can refer to any `ForeignKey` or `OneToOneField` relation in the list of fields passed to `select_related()`. This includes foreign keys that have `null=True` (which are omitted in a no-parameter `select_related()` call). It's an error to use both a list of fields and the `depth` parameter in the same `select_related()` call; they are conflicting options.

You can also refer to the reverse direction of a `OneToOneField` in the list of fields passed to `select_related` — that is, you can traverse a `OneToOneField` back to the object on which the field is defined. Instead of specifying the field name, use the `related_name` for the field on the related object.

A `OneToOneField` is not traversed in the reverse direction if you are performing a depth-based `select_related()` call.

prefetch_related

prefetch_related(**lookups*)

New in version 1.4: *Please see the release notes* Returns a `QuerySet` that will automatically retrieve, in a single batch, related objects for each of the specified lookups.

This has a similar purpose to `select_related`, in that both are designed to stop the deluge of database queries that is caused by accessing related objects, but the strategy is quite different.

`select_related` works by creating a SQL join and including the fields of the related object in the SELECT statement. For this reason, `select_related` gets the related objects in the same database query. However, to avoid the much larger result set that would result from joining across a 'many' relationship, `select_related` is limited to single-valued relationships - foreign key and one-to-one.

`prefetch_related`, on the other hand, does a separate lookup for each relationship, and does the 'joining' in Python. This allows it to prefetch many-to-many and many-to-one objects, which cannot be done

using `select_related`, in addition to the foreign key and one-to-one relationships that are supported by `select_related`. It also supports prefetching of `GenericRelation` and `GenericForeignKey`.

For example, suppose you have these models:

```
class Topping(models.Model):
    name = models.CharField(max_length=30)

class Pizza(models.Model):
    name = models.CharField(max_length=50)
    toppings = models.ManyToManyField(Topping)

    def __unicode__(self):
        return u"%s (%s)" % (self.name, u", ".join([topping.name
                                                    for topping in self.toppings.all()]))
```

and run this code:

```
>>> Pizza.objects.all()
[u"Hawaiian (ham, pineapple)", u"Seafood (prawns, smoked salmon)"...
```

The problem with this code is that it will run a query on the Toppings table for **every** item in the Pizza `QuerySet`. Using `prefetch_related`, this can be reduced to two:

```
>>> Pizza.objects.all().prefetch_related('toppings')
```

All the relevant toppings will be fetched in a single query, and used to make `QuerySets` that have a pre-filled cache of the relevant results. These `QuerySets` are then used in the `self.toppings.all()` calls.

The additional queries are executed after the QuerySet has begun to be evaluated and the primary query has been executed. Note that the result cache of the primary QuerySet and all specified related objects will then be fully loaded into memory, which is often avoided in other cases - even after a query has been executed in the database, QuerySet normally tries to make uses of chunking between the database to avoid loading all objects into memory before you need them.

Also remember that, as always with QuerySets, any subsequent chained methods which imply a different database query will ignore previously cached results, and retrieve data using a fresh database query. So, if you write the following:

```
>>> pizzas = Pizza.objects.prefetch_related('toppings')
>>> [list(pizza.toppings.filter(spicy=True)) for pizza in pizzas]
```

...then the fact that `pizza.toppings.all()` has been prefetched will not help you - in fact it hurts performance, since you have done a database query that you haven't used. So use this feature with caution!

You can also use the normal join syntax to do related fields of related fields. Suppose we have an additional model to the example above:

```
class Restaurant(models.Model):
    pizzas = models.ManyToMany(Pizza, related_name='restaurants')
    best_pizza = models.ForeignKey(Pizza, related_name='championed_by')
```

The following are all legal:

```
>>> Restaurant.objects.prefetch_related('pizzas__toppings')
```

This will prefetch all pizzas belonging to restaurants, and all toppings belonging to those pizzas. This will result in a total of 3 database queries - one for the restaurants, one for the pizzas, and one for the toppings.

```
>>> Restaurant.objects.prefetch_related('best_pizza__toppings')
```

This will fetch the best pizza and all the toppings for the best pizza for each restaurant. This will be done in 3 database queries - one for the restaurants, one for the 'best pizzas', and one for one for the toppings.

Of course, the `best_pizza` relationship could also be fetched using `select_related` to reduce the query count to 2:

```
>>> Restaurant.objects.select_related('best_pizza').prefetch_related('best_pizza__toppings')
```

Since the prefetch is executed after the main query (which includes the joins needed by `select_related`), it is able to detect that the `best_pizza` objects have already been fetched, and it will skip fetching them again.

Chaining `prefetch_related` calls will accumulate the lookups that are prefetched. To clear any `prefetch_related` behavior, pass *None* as a parameter:

```
>>> non_prefetched = qs.prefetch_related(None)
```

One difference to note when using `prefetch_related` is that objects created by a query can be shared between the different objects that they are related to i.e. a single Python model instance can appear at more than one point in the tree of objects that are returned. This will normally happen with foreign key relationships. Typically this behavior will not be a problem, and will in fact save both memory and CPU time.

While `prefetch_related` supports prefetching `GenericForeignKey` relationships, the number of queries will depend on the data. Since a `GenericForeignKey` can reference data in multiple tables, one query per table referenced is needed, rather than one query for all the items. There could be additional queries on the `ContentType` table if the relevant rows have not already been fetched.

`prefetch_related` in most cases will be implemented using a SQL query that uses the 'IN' operator. This means that for a large QuerySet a large 'IN' clause could be generated, which, depending on the database, might have performance problems of its own when it comes to parsing or executing the SQL query. Always profile for your use case!

Note that if you use `iterator()` to run the query, `prefetch_related()` calls will be ignored since these two optimizations do not make sense together.

extra

`extra` (*select=None*, *where=None*, *params=None*, *tables=None*, *order_by=None*, *select_params=None*)

Sometimes, the Django query syntax by itself can't easily express a complex `WHERE` clause. For these edge cases, Django provides the `extra() QuerySet` modifier — a hook for injecting specific clauses into the SQL generated by a `QuerySet`.

By definition, these extra lookups may not be portable to different database engines (because you're explicitly writing SQL code) and violate the DRY principle, so you should avoid them if possible.

Specify one or more of `params`, `select`, `where` or `tables`. None of the arguments is required, but you should use at least one of them.

- `select`

  The `select` argument lets you put extra fields in the `SELECT` clause. It should be a dictionary mapping attribute names to SQL clauses to use to calculate that attribute.

  Example:

  ```
  Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
  ```

  As a result, each `Entry` object will have an extra attribute, `is_recent`, a boolean representing whether the entry's `pub_date` is greater than Jan. 1, 2006.

  Django inserts the given SQL snippet directly into the `SELECT` statement, so the resulting SQL of the above example would be something like:

```
SELECT blog_entry.*, (pub_date > '2006-01-01') AS is_recent
FROM blog_entry;
```

The next example is more advanced; it does a subquery to give each resulting `Blog` object an `entry_count` attribute, an integer count of associated `Entry` objects:

```
Blog.objects.extra(
    select={
        'entry_count': 'SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id = blog_blog.id'
    },
)
```

In this particular case, we're exploiting the fact that the query will already contain the `blog_blog` table in its `FROM` clause.

The resulting SQL of the above example would be:

```
SELECT blog_blog.*, (SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id = blog_blog.id) AS
FROM blog_blog;
```

Note that the parentheses required by most database engines around subqueries are not required in Django's `select` clauses. Also note that some database backends, such as some MySQL versions, don't support subqueries.

In some rare cases, you might wish to pass parameters to the SQL fragments in `extra(select=...)`. For this purpose, use the `select_params` parameter. Since `select_params` is a sequence and the `select` attribute is a dictionary, some care is required so that the parameters are matched up correctly with the extra select pieces. In this situation, you should use a `django.utils.datastructures.SortedDict` for the `select` value, not just a normal Python dictionary.

This will work, for example:

```
Blog.objects.extra(
    select=SortedDict([('a', '%s'), ('b', '%s')]),
    select_params=('one', 'two'))
```

The only thing to be careful about when using select parameters in `extra()` is to avoid using the substring `"%%s"` (that's *two* percent characters before the `s`) in the select strings. Django's tracking of parameters looks for `%s` and an escaped `%` character like this isn't detected. That will lead to incorrect results.

- `where` / `tables`

  You can define explicit SQL `WHERE` clauses — perhaps to perform non-explicit joins — by using `where`. You can manually add tables to the SQL `FROM` clause by using `tables`.

  `where` and `tables` both take a list of strings. All `where` parameters are "AND"ed to any other search criteria.

  Example:

  ```
  Entry.objects.extra(where=["foo='a' OR bar = 'a'", "baz = 'a'"])
  ```

  ...translates (roughly) into the following SQL:

  ```
  SELECT * FROM blog_entry WHERE (foo='a' OR bar='a') AND (baz='a')
  ```

  Be careful when using the `tables` parameter if you're specifying tables that are already used in the query. When you add extra tables via the `tables` parameter, Django assumes you want that table included an extra time, if it is already included. That creates a problem, since the table name will then be given an alias. If a table appears multiple times in an SQL statement, the second and subsequent occurrences must use aliases so the database can tell them apart. If you're referring to the extra table you added in the extra `where` parameter this is going to cause errors.

Normally you'll only be adding extra tables that don't already appear in the query. However, if the case outlined above does occur, there are a few solutions. First, see if you can get by without including the extra table and use the one already in the query. If that isn't possible, put your `extra()` call at the front of the queryset construction so that your table is the first use of that table. Finally, if all else fails, look at the query produced and rewrite your `where` addition to use the alias given to your extra table. The alias will be the same each time you construct the queryset in the same way, so you can rely upon the alias name to not change.

- `order_by`

  If you need to order the resulting queryset using some of the new fields or tables you have included via `extra()` use the `order_by` parameter to `extra()` and pass in a sequence of strings. These strings should either be model fields (as in the normal `order_by()` method on querysets), of the form `table_name.column_name` or an alias for a column that you specified in the `select` parameter to `extra()`.

  For example:

  ```
  q = Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
  q = q.extra(order_by = ['-is_recent'])
  ```

  This would sort all the items for which `is_recent` is true to the front of the result set (`True` sorts before `False` in a descending ordering).

  This shows, by the way, that you can make multiple calls to `extra()` and it will behave as you expect (adding new constraints each time).

- `params`

  The `where` parameter described above may use standard Python database string placeholders — `'%s'` to indicate parameters the database engine should automatically quote. The `params` argument is a list of any extra parameters to be substituted.

  Example:

  ```
  Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
  ```

  Always use `params` instead of embedding values directly into `where` because `params` will ensure values are quoted correctly according to your particular backend. For example, quotes will be escaped correctly.

  Bad:

  ```
  Entry.objects.extra(where=["headline='Lennon'"])
  ```

  Good:

  ```
  Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
  ```

defer

`defer` (**fields*)

In some complex data-modeling situations, your models might contain a lot of fields, some of which could contain a lot of data (for example, text fields), or require expensive processing to convert them to Python objects. If you are using the results of a queryset in some situation where you know you don't know if you need those particular fields when you initially fetch the data, you can tell Django not to retrieve them from the database.

This is done by passing the names of the fields to not load to `defer()`:

```
Entry.objects.defer("headline", "body")
```

A queryset that has deferred fields will still return model instances. Each deferred field will be retrieved from the database if you access that field (one at a time, not all the deferred fields at once).

You can make multiple calls to `defer()`. Each call adds new fields to the deferred set:

```
# Defers both the body and headline fields.
Entry.objects.defer("body").filter(rating=5).defer("headline")
```

The order in which fields are added to the deferred set does not matter. Calling `defer()` with a field name that has already been deferred is harmless (the field will still be deferred).

You can defer loading of fields in related models (if the related models are loading via `select_related()`) by using the standard double-underscore notation to separate related fields:

```
Blog.objects.select_related().defer("entry__headline", "entry__body")
```

If you want to clear the set of deferred fields, pass `None` as a parameter to `defer()`:

```
# Load all fields immediately.
my_queryset.defer(None)
```

Changed in version 1.5: *Please see the release notes* Some fields in a model won't be deferred, even if you ask for them. You can never defer the loading of the primary key. If you are using `select_related()` to retrieve related models, you shouldn't defer the loading of the field that connects from the primary model to the related one, doing so will result in an error.

Note: The `defer()` method (and its cousin, `only()`, below) are only for advanced use-cases. They provide an optimization for when you have analyzed your queries closely and understand *exactly* what information you need and have measured that the difference between returning the fields you need and the full set of fields for the model will be significant.

Even if you think you are in the advanced use-case situation, **only use defer() when you cannot, at queryset load time, determine if you will need the extra fields or not**. If you are frequently loading and using a particular subset of your data, the best choice you can make is to normalize your models and put the non-loaded data into a separate model (and database table). If the columns *must* stay in the one table for some reason, create a model with `Meta.managed = False` (see the `managed attribute` documentation) containing just the fields you normally need to load and use that where you might otherwise call `defer()`. This makes your code more explicit to the reader, is slightly faster and consumes a little less memory in the Python process.

---

only

only(**fields*)

The `only()` method is more or less the opposite of `defer()`. You call it with the fields that should *not* be deferred when retrieving a model. If you have a model where almost all the fields need to be deferred, using `only()` to specify the complementary set of fields can result in simpler code.

Suppose you have a model with fields `name`, `age` and `biography`. The following two querysets are the same, in terms of deferred fields:

```
Person.objects.defer("age", "biography")
Person.objects.only("name")
```

Whenever you call `only()` it *replaces* the set of fields to load immediately. The method's name is mnemonic: **only** those fields are loaded immediately; the remainder are deferred. Thus, successive calls to `only()` result in only the final fields being considered:

```
# This will defer all fields except the headline.
Entry.objects.only("body", "rating").only("headline")
```

Since `defer()` acts incrementally (adding fields to the deferred list), you can combine calls to `only()` and `defer()` and things will behave logically:

```
# Final result is that everything except "headline" is deferred.
Entry.objects.only("headline", "body").defer("body")

# Final result loads headline and body immediately (only() replaces any
# existing set of fields).
Entry.objects.defer("body").only("headline", "body")
```

Changed in version 1.5: *Please see the release notes* All of the cautions in the note for the `defer()` documentation apply to `only()` as well. Use it cautiously and only after exhausting your other options. Also note that using `only()` and omitting a field requested using `select_related()` is an error as well.

using

`using`(*alias*)

This method is for controlling which database the `QuerySet` will be evaluated against if you are using more than one database. The only argument this method takes is the alias of a database, as defined in `DATABASES`.

For example:

```
# queries the database with the 'default' alias.
>>> Entry.objects.all()

# queries the database with the 'backup' alias
>>> Entry.objects.using('backup')
```

select_for_update

`select_for_update`(*nowait=False*)

New in version 1.4: *Please see the release notes* Returns a queryset that will lock rows until the end of the transaction, generating a `SELECT ... FOR UPDATE` SQL statement on supported databases.

For example:

```
entries = Entry.objects.select_for_update().filter(author=request.user)
```

All matched entries will be locked until the end of the transaction block, meaning that other transactions will be prevented from changing or acquiring locks on them.

Usually, if another transaction has already acquired a lock on one of the selected rows, the query will block until the lock is released. If this is not the behavior you want, call `select_for_update(nowait=True)`. This will make the call non-blocking. If a conflicting lock is already acquired by another transaction, `DatabaseError` will be raised when the queryset is evaluated.

Note that using `select_for_update()` will cause the current transaction to be considered dirty, if under transaction management. This is to ensure that Django issues a `COMMIT` or `ROLLBACK`, releasing any locks held by the `SELECT FOR UPDATE`.

Currently, the `postgresql_psycopg2`, `oracle`, and `mysql` database backends support `select_for_update()`. However, MySQL has no support for the `nowait` argument. Obviously, users of external third-party backends should check with their backend's documentation for specifics in those cases.

Passing `nowait=True` to `select_for_update` using database backends that do not support `nowait`, such as MySQL, will cause a `DatabaseError` to be raised. This is in order to prevent code unexpectedly blocking.

Using `select_for_update` on backends which do not support `SELECT ... FOR UPDATE` (such as SQLite) will have no effect.

Methods that do not return QuerySets

The following `QuerySet` methods evaluate the `QuerySet` and return something *other than* a `QuerySet`.

These methods do not use a cache (see *Caching and QuerySets*). Rather, they query the database each time they're called.

get

get(***kwargs*)

Returns the object matching the given lookup parameters, which should be in the format described in Field lookups.

`get()` raises `MultipleObjectsReturned` if more than one object was found. The `MultipleObjectsReturned` exception is an attribute of the model class.

`get()` raises a `DoesNotExist` exception if an object wasn't found for the given parameters. This exception is also an attribute of the model class. Example:

```
Entry.objects.get(id='foo') # raises Entry.DoesNotExist
```

The `DoesNotExist` exception inherits from `django.core.exceptions.ObjectDoesNotExist`, so you can target multiple `DoesNotExist` exceptions. Example:

```
from django.core.exceptions import ObjectDoesNotExist
try:
    e = Entry.objects.get(id=3)
    b = Blog.objects.get(id=1)
except ObjectDoesNotExist:
    print("Either the entry or blog doesn't exist.")
```

create

create(***kwargs*)

A convenience method for creating an object and saving it all in one step. Thus:

```
p = Person.objects.create(first_name="Bruce", last_name="Springsteen")
```

and:

```
p = Person(first_name="Bruce", last_name="Springsteen")
p.save(force_insert=True)
```

are equivalent.

The *force_insert* parameter is documented elsewhere, but all it means is that a new object will always be created. Normally you won't need to worry about this. However, if your model contains a manual primary key value that you set and if that value already exists in the database, a call to `create()` will fail with an `IntegrityError` since primary keys must be unique. Be prepared to handle the exception if you are using manual primary keys.

get_or_create

get_or_create(***kwargs*)

A convenience method for looking up an object with the given kwargs, creating one if necessary.

Returns a tuple of `(object, created)`, where `object` is the retrieved or created object and `created` is a boolean specifying whether a new object was created.

This is meant as a shortcut to boilerplatish code and is mostly useful for data-import scripts. For example:

```
try:
    obj = Person.objects.get(first_name='John', last_name='Lennon')
except Person.DoesNotExist:
    obj = Person(first_name='John', last_name='Lennon', birthday=date(1940, 10, 9))
    obj.save()
```

This pattern gets quite unwieldy as the number of fields in a model goes up. The above example can be rewritten using `get_or_create()` like so:

```
obj, created = Person.objects.get_or_create(first_name='John', last_name='Lennon',
                    defaults={'birthday': date(1940, 10, 9)})
```

Any keyword arguments passed to `get_or_create()` — *except* an optional one called `defaults` — will be used in a `get()` call. If an object is found, `get_or_create()` returns a tuple of that object and `False`. If an object is *not* found, `get_or_create()` will instantiate and save a new object, returning a tuple of the new object and `True`. The new object will be created roughly according to this algorithm:

```
defaults = kwargs.pop('defaults', {})
params = dict([(k, v) for k, v in kwargs.items() if '__' not in k])
params.update(defaults)
obj = self.model(**params)
obj.save()
```

In English, that means start with any non-`'defaults'` keyword argument that doesn't contain a double underscore (which would indicate a non-exact lookup). Then add the contents of `defaults`, overriding any keys if necessary, and use the result as the keyword arguments to the model class. As hinted at above, this is a simplification of the algorithm that is used, but it contains all the pertinent details. The internal implementation has some more error-checking than this and handles some extra edge-conditions; if you're interested, read the code.

If you have a field named `defaults` and want to use it as an exact lookup in `get_or_create()`, just use `'defaults__exact'`, like so:

```
Foo.objects.get_or_create(defaults__exact='bar', defaults={'defaults': 'baz'})
```

The `get_or_create()` method has similar error behavior to `create()` when you're using manually specified primary keys. If an object needs to be created and the key already exists in the database, an `IntegrityError` will be raised.

Finally, a word on using `get_or_create()` in Django views. As mentioned earlier, `get_or_create()` is mostly useful in scripts that need to parse data and create new records if existing ones aren't available. But if you need to use `get_or_create()` in a view, please make sure to use it only in `POST` requests unless you have a good reason not to. `GET` requests shouldn't have any effect on data; use `POST` whenever a request to a page has a side effect on your data. For more, see Safe methods in the HTTP spec.

bulk_create

bulk_create(*objs*, *batch_size=None*)

New in version 1.4: *Please see the release notes* This method inserts the provided list of objects into the database in an efficient manner (generally only 1 query, no matter how many objects there are):

```
>>> Entry.objects.bulk_create([
...     Entry(headline="Django 1.0 Released"),
...     Entry(headline="Django 1.1 Announced"),
...     Entry(headline="Breaking: Django is awesome")
... ])
```

This has a number of caveats though:

- The model's `save()` method will not be called, and the `pre_save` and `post_save` signals will not be sent.

- It does not work with child models in a multi-table inheritance scenario.
- If the model's primary key is an `AutoField` it does not retrieve and set the primary key attribute, as `save()` does.

The `batch_size` parameter controls how many objects are created in single query. The default is to create all objects in one batch, except for SQLite where the default is such that at maximum 999 variables per query is used. New in version 1.5: The `batch_size` parameter was added in version 1.5.

count

`count``()`

Returns an integer representing the number of objects in the database matching the `QuerySet`. The `count()` method never raises exceptions.

Example:

```
# Returns the total number of entries in the database.
Entry.objects.count()

# Returns the number of entries whose headline contains 'Lennon'
Entry.objects.filter(headline__contains='Lennon').count()
```

A `count()` call performs a `SELECT COUNT(*)` behind the scenes, so you should always use `count()` rather than loading all of the record into Python objects and calling `len()` on the result (unless you need to load the objects into memory anyway, in which case `len()` will be faster).

Depending on which database you're using (e.g. PostgreSQL vs. MySQL), `count()` may return a long integer instead of a normal Python integer. This is an underlying implementation quirk that shouldn't pose any real-world problems.

in_bulk

`in_bulk``(`*id_list*`)`

Takes a list of primary-key values and returns a dictionary mapping each primary-key value to an instance of the object with the given ID.

Example:

```
>>> Blog.objects.in_bulk([1])
{1: <Blog: Beatles Blog>}
>>> Blog.objects.in_bulk([1, 2])
{1: <Blog: Beatles Blog>, 2: <Blog: Cheddar Talk>}
>>> Blog.objects.in_bulk([])
{}
```

If you pass `in_bulk()` an empty list, you'll get an empty dictionary.

iterator

`iterator``()`

Evaluates the `QuerySet` (by performing the query) and returns an iterator (see **PEP 234**) over the results. A `QuerySet` typically caches its results internally so that repeated evaluations do not result in additional queries. In contrast, `iterator()` will read results directly, without doing any caching at the `QuerySet` level (internally, the default iterator calls `iterator()` and caches the return value). For a `QuerySet` which returns a large number of objects that you only need to access once, this can results in better performance and a significant reduction in memory.

Note that using `iterator()` on a `QuerySet` which has already been evaluated will force it to evaluate again, repeating the query.

Also, use of `iterator()` causes previous `prefetch_related()` calls to be ignored since these two optimizations do not make sense together.

latest

latest(*field_name=None*)

Returns the latest object in the table, by date, using the `field_name` provided as the date field.

This example returns the latest `Entry` in the table, according to the `pub_date` field:

```
Entry.objects.latest('pub_date')
```

If your model's *Meta* specifies `get_latest_by`, you can leave off the `field_name` argument to `latest()`. Django will use the field specified in `get_latest_by` by default.

Like `get()`, `latest()` raises `DoesNotExist` if there is no object with the given parameters.

Note `latest()` exists purely for convenience and readability.

aggregate

aggregate(**args*, ***kwargs*)

Returns a dictionary of aggregate values (averages, sums, etc) calculated over the `QuerySet`. Each argument to `aggregate()` specifies a value that will be included in the dictionary that is returned.

The aggregation functions that are provided by Django are described in Aggregation Functions below.

Aggregates specified using keyword arguments will use the keyword as the name for the annotation. Anonymous arguments will have a name generated for them based upon the name of the aggregate function and the model field that is being aggregated.

For example, when you are working with blog entries, you may want to know the number of authors that have contributed blog entries:

```
>>> q = Blog.objects.aggregate(Count('entry'))
{'entry__count': 16}
```

By using a keyword argument to specify the aggregate function, you can control the name of the aggregation value that is returned:

```
>>> q = Blog.objects.aggregate(number_of_entries=Count('entry'))
{'number_of_entries': 16}
```

For an in-depth discussion of aggregation, see *the topic guide on Aggregation*.

exists

exists()

Returns `True` if the `QuerySet` contains any results, and `False` if not. This tries to perform the query in the simplest and fastest way possible, but it *does* execute nearly the same query. This means that calling `QuerySet.exists()` is faster than `bool(some_query_set)`, but not by a large degree. If `some_query_set` has not yet been evaluated, but you know that it will be at some point, then using `some_query_set.exists()` will do more overall work (one query for the existence check plus an extra one to later retrieve the results) than simply using `bool(some_query_set)`, which retrieves the results and then checks if any were returned.

update

update(***kwargs*)

Performs an SQL update query for the specified fields, and returns the number of rows affected.

For example, to turn comments off for all blog entries published in 2010, you could do this:

```
>>> Entry.objects.filter(pub_date__year=2010).update(comments_on=False)
```

(This assumes your `Entry` model has fields `pub_date` and `comments_on`.)

You can update multiple fields — there's no limit on how many. For example, here we update the `comments_on` and `headline` fields:

```
>>> Entry.objects.filter(pub_date__year=2010).update(comments_on=False, headline='This is old')
```

The `update()` method is applied instantly, and the only restriction on the `QuerySet` that is updated is that it can only update columns in the model's main table, not on related models. You can't do this, for example:

```
>>> Entry.objects.update(blog__name='foo') # Won't work!
```

Filtering based on related fields is still possible, though:

```
>>> Entry.objects.filter(blog__id=1).update(comments_on=True)
```

You cannot call `update()` on a `QuerySet` that has had a slice taken or can otherwise no longer be filtered.

The `update()` method returns the number of affected rows:

```
>>> Entry.objects.filter(id=64).update(comments_on=True)
1

>>> Entry.objects.filter(slug='nonexistent-slug').update(comments_on=True)
0

>>> Entry.objects.filter(pub_date__year=2010).update(comments_on=False)
132
```

If you're just updating a record and don't need to do anything with the model object, the most efficient approach is to call `update()`, rather than loading the model object into memory. For example, instead of doing this:

```
e = Entry.objects.get(id=10)
e.comments_on = False
e.save()
```

...do this:

```
Entry.objects.filter(id=10).update(comments_on=False)
```

Using `update()` also prevents a race condition wherein something might change in your database in the short period of time between loading the object and calling `save()`.

Finally, realize that `update()` does an update at the SQL level and, thus, does not call any `save()` methods on your models, nor does it emit the `pre_save` or `post_save` signals (which are a consequence of calling `Model.save()`). If you want to update a bunch of records for a model that has a custom `save()`'() method, loop over them and call `save()`, like this:

```
for e in Entry.objects.filter(pub_date__year=2010):
    e.comments_on = False
    e.save()
```

delete

`delete``()`

Performs an SQL delete query on all rows in the `QuerySet`. The `delete()` is applied instantly. You cannot call `delete()` on a `QuerySet` that has had a slice taken or can otherwise no longer be filtered.

For example, to delete all the entries in a particular blog:

```
>>> b = Blog.objects.get(pk=1)

# Delete all the entries belonging to this Blog.
>>> Entry.objects.filter(blog=b).delete()
```

By default, Django's `ForeignKey` emulates the SQL constraint `ON DELETE CASCADE` — in other words, any objects with foreign keys pointing at the objects to be deleted will be deleted along with them. For example:

```
blogs = Blog.objects.all()
# This will delete all Blogs and all of their Entry objects.
blogs.delete()
```

New in version 1.3: This cascade behavior is customizable via the `on_delete` argument to the `ForeignKey`. The `delete()` method does a bulk delete and does not call any `delete()` methods on your models. It does, however, emit the `pre_delete` and `post_delete` signals for all deleted objects (including cascaded deletions).

Field lookups

Field lookups are how you specify the meat of an SQL `WHERE` clause. They're specified as keyword arguments to the `QuerySet` methods `filter()`, `exclude()` and `get()`.

For an introduction, see *models and database queries documentation*.

exact Exact match. If the value provided for comparison is `None`, it will be interpreted as an SQL `NULL` (see `isnull` for more details).

Examples:

```
Entry.objects.get(id__exact=14)
Entry.objects.get(id__exact=None)
```

SQL equivalents:

```
SELECT ... WHERE id = 14;
SELECT ... WHERE id IS NULL;
```

MySQL comparisons

In MySQL, a database table's "collation" setting determines whether `exact` comparisons are case-sensitive. This is a database setting, *not* a Django setting. It's possible to configure your MySQL tables to use case-sensitive comparisons, but some trade-offs are involved. For more information about this, see the *collation section* in the *databases* documentation.

---

iexact Case-insensitive exact match.

Example:

```
Blog.objects.get(name__iexact='beatles blog')
```

SQL equivalent:

```
SELECT ... WHERE name ILIKE 'beatles blog';
```

Note this will match `'Beatles Blog'`, `'beatles blog'`, `'BeAtLes BLoG'`, etc.

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons. SQLite does not do case-insensitive matching for Unicode strings.

---

contains Case-sensitive containment test.

Example:

```
Entry.objects.get(headline__contains='Lennon')
```

SQL equivalent:

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

Note this will match the headline `'Lennon honored today'` but not `'lennon honored today'`.

SQLite users

SQLite doesn't support case-sensitive `LIKE` statements; `contains` acts like `icontains` for SQLite. See the *database note* for more information.

---

icontains Case-insensitive containment test.

Example:

```
Entry.objects.get(headline__icontains='Lennon')
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE '%Lennon%';
```

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons.

---

in In a given list.

Example:

```
Entry.objects.filter(id__in=[1, 3, 4])
```

SQL equivalent:

```
SELECT ... WHERE id IN (1, 3, 4);
```

You can also use a queryset to dynamically evaluate the list of values instead of providing a list of literal values:

```
inner_qs = Blog.objects.filter(name__contains='Cheddar')
entries = Entry.objects.filter(blog__in=inner_qs)
```

This queryset will be evaluated as subselect statement:

```
SELECT ... WHERE blog.id IN (SELECT id FROM ... WHERE NAME LIKE '%Cheddar%')
```

If you pass in a `ValuesQuerySet` or `ValuesListQuerySet` (the result of calling `values()` or `values_list()` on a queryset) as the value to an `__in` lookup, you need to ensure you are only extracting one field in the result. For example, this will work (filtering on the blog names):

```
inner_qs = Blog.objects.filter(name__contains='Ch').values('name')
entries = Entry.objects.filter(blog__name__in=inner_qs)
```

This example will raise an exception, since the inner query is trying to extract two field values, where only one is expected:

```
# Bad code! Will raise a TypeError.
inner_qs = Blog.objects.filter(name__contains='Ch').values('name', 'id')
entries = Entry.objects.filter(blog__name__in=inner_qs)
```

Performance considerations

Be cautious about using nested queries and understand your database server's performance characteristics (if in doubt, benchmark!). Some database backends, most notably MySQL, don't optimize nested queries very well. It is more efficient, in those cases, to extract a list of values and then pass that into the second query. That is, execute two queries instead of one:

```
values = Blog.objects.filter(
        name__contains='Cheddar').values_list('pk', flat=True)
entries = Entry.objects.filter(blog__in=list(values))
```

Note the `list()` call around the Blog `QuerySet` to force execution of the first query. Without it, a nested query would be executed, because *QuerySets are lazy*.

---

gt Greater than.

Example:

```
Entry.objects.filter(id__gt=4)
```

SQL equivalent:

```
SELECT ... WHERE id > 4;
```

gte Greater than or equal to.

lt Less than.

lte Less than or equal to.

startswith Case-sensitive starts-with.

Example:

```
Entry.objects.filter(headline__startswith='Will')
```

SQL equivalent:

```
SELECT ... WHERE headline LIKE 'Will%';
```

SQLite doesn't support case-sensitive `LIKE` statements; `startswith` acts like `istartswith` for SQLite.

istartswith Case-insensitive starts-with.

Example:

```
Entry.objects.filter(headline__istartswith='will')
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE 'Will%';
```

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons.

---

endswith Case-sensitive ends-with.

Example:

```
Entry.objects.filter(headline__endswith='cats')
```

SQL equivalent:

```
SELECT ... WHERE headline LIKE '%cats';
```

SQLite users

SQLite doesn't support case-sensitive `LIKE` statements; `endswith` acts like `iendswith` for SQLite. Refer to the *database note* documentation for more.

---

iendswith Case-insensitive ends-with.

Example:

```
Entry.objects.filter(headline__iendswith='will')
```

SQL equivalent:

```
SELECT ... WHERE headline ILIKE '%will'
```

SQLite users

When using the SQLite backend and Unicode (non-ASCII) strings, bear in mind the *database note* about string comparisons.

---

range Range test (inclusive).

Example:

```
start_date = datetime.date(2005, 1, 1)
end_date = datetime.date(2005, 3, 31)
Entry.objects.filter(pub_date__range=(start_date, end_date))
```

SQL equivalent:

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' and '2005-03-31';
```

You can use `range` anywhere you can use `BETWEEN` in SQL — for dates, numbers and even characters.

year For date/datetime fields, exact year match. Takes a four-digit year.

Example:

```
Entry.objects.filter(pub_date__year=2005)
```

SQL equivalent:

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' AND '2005-12-31 23:59:59.999999';
```

(The exact SQL syntax varies for each database engine.)

month For date and datetime fields, an exact month match. Takes an integer 1 (January) through 12 (December).

Example:

```
Entry.objects.filter(pub_date__month=12)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('month' FROM pub_date) = '12';
```

(The exact SQL syntax varies for each database engine.)

day For date and datetime fields, an exact day match.

Example:

```
Entry.objects.filter(pub_date__day=3)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('day' FROM pub_date) = '3';
```

(The exact SQL syntax varies for each database engine.)

Note this will match any record with a pub_date on the third day of the month, such as January 3, July 3, etc.

week_day For date and datetime fields, a 'day of the week' match.

Takes an integer value representing the day of week from 1 (Sunday) to 7 (Saturday).

Example:

```
Entry.objects.filter(pub_date__week_day=2)
```

(No equivalent SQL code fragment is included for this lookup because implementation of the relevant query varies among different database engines.)

Note this will match any record with a `pub_date` that falls on a Monday (day 2 of the week), regardless of the month or year in which it occurs. Week days are indexed with day 1 being Sunday and day 7 being Saturday.

Warning: When *time zone support* is enabled, Django uses UTC in the database connection, which means the `year`, `month`, `day` and `week_day` lookups are performed in UTC. This is a known limitation of the current implementation.

isnull Takes either `True` or `False`, which correspond to SQL queries of `IS NULL` and `IS NOT NULL`, respectively.

Example:

```
Entry.objects.filter(pub_date__isnull=True)
```

SQL equivalent:

```
SELECT ... WHERE pub_date IS NULL;
```

search A boolean full-text search, taking advantage of full-text indexing. This is like `contains` but is significantly faster due to full-text indexing.

Example:

```
Entry.objects.filter(headline__search="+Django -jazz Python")
```

SQL equivalent:

```
SELECT ... WHERE MATCH(tablename, headline) AGAINST (+Django -jazz Python IN BOOLEAN MODE);
```

Note this is only available in MySQL and requires direct manipulation of the database to add the full-text index. By default Django uses BOOLEAN MODE for full text searches. See the MySQL documentation for additional details.

regex Case-sensitive regular expression match.

The regular expression syntax is that of the database backend in use. In the case of SQLite, which has no built in regular expression support, this feature is provided by a (Python) user-defined REGEXP function, and the regular expression syntax is therefore that of Python's `re` module.

Example:

```
Entry.objects.get(title__regex=r'^(An?|The) +')
```

SQL equivalents:

```
SELECT ... WHERE title REGEXP BINARY '^(An?|The) +'; -- MySQL

SELECT ... WHERE REGEXP_LIKE(title, '^(an?|the) +', 'c'); -- Oracle

SELECT ... WHERE title ~ '^(An?|The) +'; -- PostgreSQL

SELECT ... WHERE title REGEXP '^(An?|The) +'; -- SQLite
```

Using raw strings (e.g., `r'foo'` instead of `'foo'`) for passing in the regular expression syntax is recommended.

iregex Case-insensitive regular expression match.

Example:

```
Entry.objects.get(title__iregex=r'^(an?|the) +')
```

SQL equivalents:

```
SELECT ... WHERE title REGEXP '^(an?|the) +'; -- MySQL

SELECT ... WHERE REGEXP_LIKE(title, '^(an?|the) +', 'i'); -- Oracle
```

```
SELECT ... WHERE title ~* '^(an?|the) +'; -- PostgreSQL

SELECT ... WHERE title REGEXP '(?i)^(an?|the) +'; -- SQLite
```

Aggregation functions

Django provides the following aggregation functions in the `django.db.models` module. For details on how to use these aggregate functions, see *the topic guide on aggregation*.

Avg

class **Avg** (*field*)

Returns the mean value of the given field, which must be numeric.

- Default alias: `<field>__avg`
- Return type: `float`

Count

class **Count** (*field*, *distinct=False*)

Returns the number of objects that are related through the provided field.

- Default alias: `<field>__count`
- Return type: `int`

Has one optional argument:

distinct

If `distinct=True`, the count will only include unique instances. This is the SQL equivalent of `COUNT(DISTINCT <field>)`. The default value is `False`.

Max

class **Max** (*field*)

Returns the maximum value of the given field.

- Default alias: `<field>__max`
- Return type: same as input field

Min

class **Min** (*field*)

Returns the minimum value of the given field.

- Default alias: `<field>__min`
- Return type: same as input field

StdDev

class **StdDev** (*field*, *sample=False*)

Returns the standard deviation of the data in the provided field.

- Default alias: `<field>__stddev`
- Return type: `float`

Has one optional argument:

sample
By default, `StdDev` returns the population standard deviation. However, if `sample=True`, the return value will be the sample standard deviation.

SQLite

SQLite doesn't provide `StdDev` out of the box. An implementation is available as an extension module for SQLite. Consult the SQlite documentation for instructions on obtaining and installing this extension.

Sum

class Sum(*field*)

Computes the sum of all values of the given field.

- Default alias: `<field>__sum`
- Return type: same as input field

Variance

class Variance(*field*, *sample=False*)

Returns the variance of the data in the provided field.

- Default alias: `<field>__variance`
- Return type: `float`

Has one optional argument:

sample
By default, `Variance` returns the population variance. However, if `sample=True`, the return value will be the sample variance.

SQLite

SQLite doesn't provide `Variance` out of the box. An implementation is available as an extension module for SQLite. Consult the SQlite documentation for instructions on obtaining and installing this extension.

6.13 Request and response objects

6.13.1 Quick overview

Django uses request and response objects to pass state through the system.

When a page is requested, Django creates an `HttpRequest` object that contains metadata about the request. Then Django loads the appropriate view, passing the `HttpRequest` as the first argument to the view function. Each view is responsible for returning an `HttpResponse` object.

This document explains the APIs for `HttpRequest` and `HttpResponse` objects.

6.13.2 HttpRequest objects

class HttpRequest

Attributes

All attributes should be considered read-only, unless stated otherwise below. `session` is a notable exception.

`HttpRequest.`**`body`**

Changed in version 1.4: *Please see the release notes* Before Django 1.4, `HttpRequest.body` was named `HttpRequest.raw_post_data`.

The raw HTTP request body as a byte string. This is useful for processing data in different ways than conventional HTML forms: binary images, XML payload etc. For processing conventional form data, use `HttpRequest.POST`. New in version 1.3: *Please see the release notes* You can also read from an HttpRequest using a file-like interface. See `HttpRequest.read()`.

`HttpRequest.`**`path`**

A string representing the full path to the requested page, not including the domain.

Example: `"/music/bands/the_beatles/"`

`HttpRequest.`**`path_info`**

Under some Web server configurations, the portion of the URL after the host name is split up into a script prefix portion and a path info portion. The `path_info` attribute always contains the path info portion of the path, no matter what Web server is being used. Using this instead of attr:*~HttpRequest.path* can make your code much easier to move between test and deployment servers.

For example, if the `WSGIScriptAlias` for your application is set to `"/minfo"`, then `path` might be `"/minfo/music/bands/the_beatles/"` and `path_info` would be `"/music/bands/the_beatles/"`.

`HttpRequest.`**`method`**

A string representing the HTTP method used in the request. This is guaranteed to be uppercase. Example:

```
if request.method == 'GET':
    do_something()
elif request.method == 'POST':
    do_something_else()
```

`HttpRequest.`**`encoding`**

A string representing the current encoding used to decode form submission data (or `None`, which means the `DEFAULT_CHARSET` setting is used). You can write to this attribute to change the encoding used when accessing the form data. Any subsequent attribute accesses (such as reading from `GET` or `POST`) will use the new `encoding` value. Useful if you know the form data is not in the `DEFAULT_CHARSET` encoding.

`HttpRequest.`**`GET`**

A dictionary-like object containing all given HTTP GET parameters. See the `QueryDict` documentation below.

`HttpRequest.`**`POST`**

A dictionary-like object containing all given HTTP POST parameters. See the `QueryDict` documentation below.

It's possible that a request can come in via POST with an empty `POST` dictionary – if, say, a form is requested via the POST HTTP method but does not include form data. Therefore, you shouldn't use `if request.POST` to check for use of the POST method; instead, use `if request.method == "POST"` (see above).

Note: `POST` does *not* include file-upload information. See `FILES`.

`HttpRequest.`**`REQUEST`**

For convenience, a dictionary-like object that searches `POST` first, then `GET`. Inspired by PHP's `$_REQUEST`.

For example, if `GET = {"name": "john"}` and `POST = {"age": '34'}`, `REQUEST["name"]` would be `"john"`, and `REQUEST["age"]` would be `"34"`.

It's strongly suggested that you use `GET` and `POST` instead of `REQUEST`, because the former are more explicit.

`HttpRequest.`**`COOKIES`**
A standard Python dictionary containing all cookies. Keys and values are strings.

`HttpRequest.`**`FILES`**
A dictionary-like object containing all uploaded files. Each key in `FILES` is the `name` from the `<input type="file" name="" />`. Each value in `FILES` is an `UploadedFile` as described below.

See *Managing files* for more information.

Note that `FILES` will only contain data if the request method was POST and the `<form>` that posted to the request had `enctype="multipart/form-data"`. Otherwise, `FILES` will be a blank dictionary-like object.

`HttpRequest.`**`META`**
A standard Python dictionary containing all available HTTP headers. Available headers depend on the client and server, but here are some examples:

- `CONTENT_LENGTH` – the length of the request body (as a string).
- `CONTENT_TYPE` – the MIME type of the request body.
- `HTTP_ACCEPT_ENCODING` – Acceptable encodings for the response.
- `HTTP_ACCEPT_LANGUAGE` – Acceptable languages for the response.
- `HTTP_HOST` – The HTTP Host header sent by the client.
- `HTTP_REFERER` – The referring page, if any.
- `HTTP_USER_AGENT` – The client's user-agent string.
- `QUERY_STRING` – The query string, as a single (unparsed) string.
- `REMOTE_ADDR` – The IP address of the client.
- `REMOTE_HOST` – The hostname of the client.
- `REMOTE_USER` – The user authenticated by the Web server, if any.
- `REQUEST_METHOD` – A string such as `"GET"` or `"POST"`.
- `SERVER_NAME` – The hostname of the server.
- `SERVER_PORT` – The port of the server (as a string).

With the exception of `CONTENT_LENGTH` and `CONTENT_TYPE`, as given above, any HTTP headers in the request are converted to `META` keys by converting all characters to uppercase, replacing any hyphens with underscores and adding an `HTTP_` prefix to the name. So, for example, a header called `X-Bender` would be mapped to the `META` key `HTTP_X_BENDER`.

`HttpRequest.`**`user`**
A `django.contrib.auth.models.User` object representing the currently logged-in user. If the user isn't currently logged in, `user` will be set to an instance of `django.contrib.auth.models.AnonymousUser`. You can tell them apart with `is_authenticated()`, like so:

```
if request.user.is_authenticated():
    # Do something for logged-in users.
else:
    # Do something for anonymous users.
```

`user` is only available if your Django installation has the `AuthenticationMiddleware` activated. For more, see *User authentication in Django*.

HttpRequest.**session**
A readable-and-writable, dictionary-like object that represents the current session. This is only available if your Django installation has session support activated. See the *session documentation* for full details.

HttpRequest.**urlconf**
Not defined by Django itself, but will be read if other code (e.g., a custom middleware class) sets it. When present, this will be used as the root URLconf for the current request, overriding the ROOT_URLCONF setting. See *How Django processes a request* for details.

Methods

HttpRequest.**get_host**()
Returns the originating host of the request using information from the HTTP_X_FORWARDED_HOST (if USE_X_FORWARDED_HOST is enabled) and HTTP_HOST headers, in that order. If they don't provide a value, the method uses a combination of SERVER_NAME and SERVER_PORT as detailed in **PEP 3333**.

Example: "127.0.0.1:8000"

Note: The get_host() method fails when the host is behind multiple proxies. One solution is to use middleware to rewrite the proxy headers, as in the following example:

```
class MultipleProxyMiddleware(object):
    FORWARDED_FOR_FIELDS = [
        'HTTP_X_FORWARDED_FOR',
        'HTTP_X_FORWARDED_HOST',
        'HTTP_X_FORWARDED_SERVER',
    ]

    def process_request(self, request):
        """
        Rewrites the proxy headers so that only the most
        recent proxy is used.
        """
        for field in self.FORWARDED_FOR_FIELDS:
            if field in request.META:
                if ',' in request.META[field]:
                    parts = request.META[field].split(',')
                    request.META[field] = parts[-1].strip()
```

This middleware should be positioned before any other middleware that relies on the value of get_host() – for instance, CommonMiddleware or CsrfViewMiddleware.

HttpRequest.**get_full_path**()
Returns the path, plus an appended query string, if applicable.

Example: "/music/bands/the_beatles/?print=true"

HttpRequest.**build_absolute_uri**(*location*)
Returns the absolute URI form of location. If no location is provided, the location will be set to request.get_full_path().

If the location is already an absolute URI, it will not be altered. Otherwise the absolute URI is built using the server variables available in this request.

Example: "http://example.com/music/bands/the_beatles/?print=true"

HttpRequest.**get_signed_cookie**(*key*, *default=RAISE_ERROR*, *salt=''*, *max_age=None*)
New in version 1.4: *Please see the release notes* Returns a cookie value for a signed cookie, or raises a

`BadSignature` exception if the signature is no longer valid. If you provide the `default` argument the exception will be suppressed and that default value will be returned instead.

The optional `salt` argument can be used to provide extra protection against brute force attacks on your secret key. If supplied, the `max_age` argument will be checked against the signed timestamp attached to the cookie value to ensure the cookie is not older than `max_age` seconds.

For example:

```
>>> request.get_signed_cookie('name')
'Tony'
>>> request.get_signed_cookie('name', salt='name-salt')
'Tony' # assuming cookie was set using the same salt
>>> request.get_signed_cookie('non-existing-cookie')
...
KeyError: 'non-existing-cookie'
>>> request.get_signed_cookie('non-existing-cookie', False)
False
>>> request.get_signed_cookie('cookie-that-was-tampered-with')
...
BadSignature: ...
>>> request.get_signed_cookie('name', max_age=60)
...
SignatureExpired: Signature age 1677.3839159 > 60 seconds
>>> request.get_signed_cookie('name', False, max_age=60)
False
```

See *cryptographic signing* for more information.

`HttpRequest.`**`is_secure`**`()`
Returns `True` if the request is secure; that is, if it was made with HTTPS.

`HttpRequest.`**`is_ajax`**`()`
Returns `True` if the request was made via an `XMLHttpRequest`, by checking the `HTTP_X_REQUESTED_WITH` header for the string `'XMLHttpRequest'`. Most modern JavaScript libraries send this header. If you write your own XMLHttpRequest call (on the browser side), you'll have to set this header manually if you want `is_ajax()` to work.

`HttpRequest.`**`read`**`(`*size=None*`)`

`HttpRequest.`**`readline`**`()`

`HttpRequest.`**`readlines`**`()`

`HttpRequest.`**`xreadlines`**`()`

`HttpRequest.`**`__iter__`**`()`
New in version 1.3: *Please see the release notes* Methods implementing a file-like interface for reading from an HttpRequest instance. This makes it possible to consume an incoming request in a streaming fashion. A common use-case would be to process a big XML payload with iterative parser without constructing a whole XML tree in memory.

Given this standard interface, an HttpRequest instance can be passed directly to an XML parser such as ElementTree:

```
import xml.etree.ElementTree as ET
for element in ET.iterparse(request):
    process(element)
```

6.13.3 UploadedFile objects

class `UploadedFile`

Attributes

`UploadedFile.`**`name`**
The name of the uploaded file.

`UploadedFile.`**`size`**
The size, in bytes, of the uploaded file.

Methods

`UploadedFile.`**`chunks`**(*chunk_size=None*)
Returns a generator that yields sequential chunks of data.

`UploadedFile.`**`read`**(*num_bytes=None*)
Read a number of bytes from the file.

6.13.4 QueryDict objects

class `QueryDict`

In an `HttpRequest` object, the `GET` and `POST` attributes are instances of `django.http.QueryDict`. `QueryDict` is a dictionary-like class customized to deal with multiple values for the same key. This is necessary because some HTML form elements, notably `<select multiple="multiple">`, pass multiple values for the same key.

`QueryDict` instances are immutable, unless you create a `copy()` of them. That means you can't change attributes of `request.POST` and `request.GET` directly.

Methods

`QueryDict` implements all the standard dictionary methods, because it's a subclass of dictionary. Exceptions are outlined here:

`QueryDict.`**`__getitem__`**(*key*)
Returns the value for the given key. If the key has more than one value, `__getitem__()` returns the last value. Raises `django.utils.datastructures.MultiValueDictKeyError` if the key does not exist. (This is a subclass of Python's standard `KeyError`, so you can stick to catching `KeyError`.)

`QueryDict.`**`__setitem__`**(*key*, *value*)
Sets the given key to `[value]` (a Python list whose single element is `value`). Note that this, as other dictionary functions that have side effects, can only be called on a mutable `QueryDict` (one that was created via `copy()`).

`QueryDict.`**`__contains__`**(*key*)
Returns `True` if the given key is set. This lets you do, e.g., `if "foo" in request.GET`.

`QueryDict.`**`get`**(*key*, *default*)
Uses the same logic as `__getitem__()` above, with a hook for returning a default value if the key doesn't exist.

QueryDict.**setdefault**(*key*, *default*)
Just like the standard dictionary `setdefault()` method, except it uses `__setitem__()` internally.

QueryDict.**update**(*other_dict*)
Takes either a `QueryDict` or standard dictionary. Just like the standard dictionary `update()` method, except it *appends* to the current dictionary items rather than replacing them. For example:

```
>>> q = QueryDict('a=1')
>>> q = q.copy() # to make it mutable
>>> q.update({'a': '2'})
>>> q.getlist('a')
[u'1', u'2']
>>> q['a'] # returns the last
[u'2']
```

QueryDict.**items**()
Just like the standard dictionary `items()` method, except this uses the same last-value logic as `__getitem__()`. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[(u'a', u'3')]
```

QueryDict.**iteritems**()
Just like the standard dictionary `iteritems()` method. Like `QueryDict.items()` this uses the same last-value logic as `QueryDict.__getitem__()`.

QueryDict.**iterlists**()
Like `QueryDict.iteritems()` except it includes all values, as a list, for each member of the dictionary.

QueryDict.**values**()
Just like the standard dictionary `values()` method, except this uses the same last-value logic as `__getitem__()`. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
[u'3']
```

QueryDict.**itervalues**()
Just like `QueryDict.values()`, except an iterator.

In addition, `QueryDict` has the following methods:

QueryDict.**copy**()
Returns a copy of the object, using `copy.deepcopy()` from the Python standard library. The copy will be mutable – that is, you can change its values.

QueryDict.**getlist**(*key*, *default*)
Returns the data with the requested key, as a Python list. Returns an empty list if the key doesn't exist and no default value was provided. It's guaranteed to return a list of some sort unless the default value was no list. Changed in version 1.4: The `default` parameter was added.

QueryDict.**setlist**(*key*, *list_*)
Sets the given key to `list_` (unlike `__setitem__()`).

QueryDict.**appendlist**(*key*, *item*)
Appends an item to the internal list associated with key.

QueryDict.**setlistdefault**(*key*, *default_list*)
Just like `setdefault`, except it takes a list of values instead of a single value.

QueryDict.**lists**()
Like items(), except it includes all values, as a list, for each member of the dictionary. For example:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[(u'a', [u'1', u'2', u'3'])]
```

QueryDict.**dict**()
New in version 1.4: *Please see the release notes* Returns dict representation of QueryDict. For every (key, list) pair in QueryDict, dict will have (key, item), where item is one element of the list, using same logic as QueryDict.__getitem__():

```
>>> q = QueryDict('a=1&a=3&a=5')
>>> q.dict()
{u'a': u'5'}
```

QueryDict.**urlencode**([*safe*])
Returns a string of the data in query-string format. Example:

```
>>> q = QueryDict('a=2&b=3&b=5')
>>> q.urlencode()
'a=2&b=3&b=5'
```

Changed in version 1.3: The safe parameter was added. Optionally, urlencode can be passed characters which do not require encoding. For example:

```
>>> q = QueryDict('', mutable=True)
>>> q['next'] = '/a&b/'
>>> q.urlencode(safe='/')
'next=/a%26b/'
```

6.13.5 HttpResponse objects

class HttpResponse

In contrast to HttpRequest objects, which are created automatically by Django, HttpResponse objects are your responsibility. Each view you write is responsible for instantiating, populating and returning an HttpResponse.

The HttpResponse class lives in the django.http module.

Usage

Passing strings

Typical usage is to pass the contents of the page, as a string, to the HttpResponse constructor:

```
>>> from django.http import HttpResponse
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", content_type="text/plain")
```

But if you want to add content incrementally, you can use response as a file-like object:

```
>>> response = HttpResponse()
>>> response.write("<p>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</p>")
```

Passing iterators

Finally, you can pass `HttpResponse` an iterator rather than passing it hard-coded strings. If you use this technique, follow these guidelines:

- The iterator should return strings.
- If an `HttpResponse` has been initialized with an iterator as its content, you can't use the `HttpResponse` instance as a file-like object. Doing so will raise `Exception`.

Setting headers

To set or remove a header in your response, treat it like a dictionary:

```
>>> response = HttpResponse()
>>> response['Cache-Control'] = 'no-cache'
>>> del response['Cache-Control']
```

Note that unlike a dictionary, `del` doesn't raise `KeyError` if the header doesn't exist.

HTTP headers cannot contain newlines. An attempt to set a header containing a newline character (CR or LF) will raise `BadHeaderError`

Telling the browser to treat the response as a file attachment

To tell the browser to treat the response as a file attachment, use the `content_type` argument and set the `Content-Disposition` header. For example, this is how you might return a Microsoft Excel spreadsheet:

```
>>> response = HttpResponse(my_data, content_type='application/vnd.ms-excel')
>>> response['Content-Disposition'] = 'attachment; filename=foo.xls'
```

There's nothing Django-specific about the `Content-Disposition` header, but it's easy to forget the syntax, so we've included it here.

Attributes

`HttpResponse.`**`content`**
A string representing the content, encoded from a Unicode object if necessary.

`HttpResponse.`**`status_code`**
The HTTP Status code for the response.

Methods

`HttpResponse.`**`__init__`**(*content=''*, *content_type=None*, *status=200*)
Instantiates an `HttpResponse` object with the given page content and content type.

`content` should be an iterator or a string. If it's an iterator, it should return strings, and those strings will be joined together to form the content of the response. If it is not an iterator or a string, it will be converted to a string when accessed.

`content_type` is the MIME type optionally completed by a character set encoding and is used to fill the HTTP `Content-Type` header. If not specified, it is formed by the `DEFAULT_CONTENT_TYPE` and `DEFAULT_CHARSET` settings, by default: "*text/html; charset=utf-8*".

Historically, this parameter was called `mimetype` (now deprecated).

`status` is the HTTP Status code for the response.

`HttpResponse.__setitem__`(*header*, *value*)
Sets the given header name to the given value. Both `header` and `value` should be strings.

`HttpResponse.__delitem__`(*header*)
Deletes the header with the given name. Fails silently if the header doesn't exist. Case-insensitive.

`HttpResponse.__getitem__`(*header*)
Returns the value for the given header name. Case-insensitive.

`HttpResponse.has_header`(*header*)
Returns `True` or `False` based on a case-insensitive check for a header with the given name.

`HttpResponse.set_cookie`(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=True*)
Changed in version 1.3: *Please see the release notes* The possibility of specifying a `datetime.datetime` object in `expires`, and the auto-calculation of `max_age` in such case was added. The `httponly` argument was also added. Changed in version 1.4: *Please see the release notes* The default value for httponly was changed from `False` to `True`.

Sets a cookie. The parameters are the same as in the `Cookie.Morsel` object in the Python standard library.

- `max_age` should be a number of seconds, or `None` (default) if the cookie should last only as long as the client's browser session. If `expires` is not specified, it will be calculated.
- `expires` should either be a string in the format `"Wdy, DD-Mon-YY HH:MM:SS GMT"` or a `datetime.datetime` object in UTC. If `expires` is a `datetime` object, the `max_age` will be calculated.
- Use `domain` if you want to set a cross-domain cookie. For example, `domain=".lawrence.com"` will set a cookie that is readable by the domains www.lawrence.com, blogs.lawrence.com and calendars.lawrence.com. Otherwise, a cookie will only be readable by the domain that set it.
- Use `httponly=True` if you want to prevent client-side JavaScript from having access to the cookie.

  HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the **RFC 2109** standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of client side script accessing the protected cookie data.

`HttpResponse.set_signed_cookie`(*key*, *value=''*, *salt=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=None*, *httponly=True*)
New in version 1.4: *Please see the release notes* Like `set_cookie()`, but *cryptographic signing* the cookie before setting it. Use in conjunction with `HttpRequest.get_signed_cookie()`. You can use the optional `salt` argument for added key strength, but you will need to remember to pass it to the corresponding `HttpRequest.get_signed_cookie()` call.

`HttpResponse.delete_cookie`(*key*, *path='/'*, *domain=None*)
Deletes the cookie with the given key. Fails silently if the key doesn't exist.

Due to the way cookies work, `path` and `domain` should be the same values you used in `set_cookie()` – otherwise the cookie may not be deleted.

`HttpResponse.write`(*content*)
This method makes an `HttpResponse` instance a file-like object.

`HttpResponse.flush`()
This method makes an `HttpResponse` instance a file-like object.

`HttpResponse.tell`()
This method makes an `HttpResponse` instance a file-like object.

HttpResponse subclasses

Django includes a number of `HttpResponse` subclasses that handle different types of HTTP responses. Like `HttpResponse`, these subclasses live in `django.http`.

class HttpResponseRedirect
The constructor takes a single argument – the path to redirect to. This can be a fully qualified URL (e.g. `'http://www.yahoo.com/search/'`) or an absolute path with no domain (e.g. `'/search/'`). Note that this returns an HTTP status code 302.

class HttpResponsePermanentRedirect
Like `HttpResponseRedirect`, but it returns a permanent redirect (HTTP status code 301) instead of a "found" redirect (status code 302).

class HttpResponseNotModified
The constructor doesn't take any arguments. Use this to designate that a page hasn't been modified since the user's last request (status code 304).

class HttpResponseBadRequest
Acts just like `HttpResponse` but uses a 400 status code.

class HttpResponseNotFound
Acts just like `HttpResponse` but uses a 404 status code.

class HttpResponseForbidden
Acts just like `HttpResponse` but uses a 403 status code.

class HttpResponseNotAllowed
Like `HttpResponse`, but uses a 405 status code. Takes a single, required argument: a list of permitted methods (e.g. `['GET', 'POST']`).

class HttpResponseGone
Acts just like `HttpResponse` but uses a 410 status code.

class HttpResponseServerError
Acts just like `HttpResponse` but uses a 500 status code.

Note: If a custom subclass of `HttpResponse` implements a `render` method, Django will treat it as emulating a `SimpleTemplateResponse`, and the `render` method must itself return a valid response object.

6.14 TemplateResponse and SimpleTemplateResponse

New in version 1.3: *Please see the release notes* Standard `HttpResponse` objects are static structures. They are provided with a block of pre-rendered content at time of construction, and while that content can be modified, it isn't in a form that makes it easy to perform modifications.

However, it can sometimes be beneficial to allow decorators or middleware to modify a response *after* it has been constructed by the view. For example, you may want to change the template that is used, or put additional data into the context.

TemplateResponse provides a way to do just that. Unlike basic `HttpResponse` objects, TemplateResponse objects retain the details of the template and context that was provided by the view to compute the response. The final output of the response is not computed until it is needed, later in the response process.

6.14.1 SimpleTemplateResponse objects

class SimpleTemplateResponse

Attributes

SimpleTemplateResponse.**template_name**
The name of the template to be rendered. Accepts a Template object, a path to a template or list of template paths.

Example: ['foo.html', 'path/to/bar.html']

SimpleTemplateResponse.**context_data**
The context data to be used when rendering the template. It can be a dictionary or a context object.

Example: {'foo': 123}

SimpleTemplateResponse.**rendered_content**
The current rendered value of the response content, using the current template and context data.

SimpleTemplateResponse.**is_rendered**
A boolean indicating whether the response content has been rendered.

Methods

SimpleTemplateResponse.**__init__**(*template*, *context=None*, *mimetype=None*, *status=None*, *content_type=None*)
Instantiates a SimpleTemplateResponse object with the given template, context, MIME type and HTTP status.

template The full name of a template, or a sequence of template names. Template instances can also be used.

context A dictionary of values to add to the template context. By default, this is an empty dictionary. Context objects are also accepted as context values.

status The HTTP Status code for the response.

content_type An alias for mimetype. Historically, this parameter was only called mimetype, but since this is actually the value included in the HTTP Content-Type header, it can also include the character set encoding, which makes it more than just a MIME type specification. If mimetype is specified (not None), that value is used. Otherwise, content_type is used. If neither is given, DEFAULT_CONTENT_TYPE is used.

SimpleTemplateResponse.**resolve_context**(*context*)
Converts context data into a context instance that can be used for rendering a template. Accepts a dictionary of context data or a context object. Returns a Context instance containing the provided data.

Override this method in order to customize context instantiation.

SimpleTemplateResponse.**resolve_template**(*template*)
Resolves the template instance to use for rendering. Accepts a path of a template to use, or a sequence of template paths. Template instances may also be provided. Returns the Template instance to be rendered.

Override this method in order to customize template rendering.

SimpleTemplateResponse.**add_post_render_callback**()
Add a callback that will be invoked after rendering has taken place. This hook can be used to defer certain processing operations (such as caching) until after rendering has occurred.

If the `SimpleTemplateResponse` has already been rendered, the callback will be invoked immediately.

When called, callbacks will be passed a single argument – the rendered `SimpleTemplateResponse` instance.

If the callback returns a value that is not *None*, this will be used as the response instead of the original response object (and will be passed to the next post rendering callback etc.)

`SimpleTemplateResponse.render()`:
Sets `response.content` to the result obtained by `SimpleTemplateResponse.rendered_content`, runs all post-rendering callbacks, and returns the resulting response object.

`render()` will only have an effect the first time it is called. On subsequent calls, it will return the result obtained from the first call.

6.14.2 TemplateResponse objects

class `TemplateResponse`
TemplateResponse is a subclass of `SimpleTemplateResponse` that uses a `RequestContext` instead of a `Context`.

Methods

`TemplateResponse.`**`__init__`**(*request*, *template*, *context=None*, *mimetype=None*, *status=None*, *content_type=None*, *current_app=None*)
Instantiates an `TemplateResponse` object with the given template, context, MIME type and HTTP status.

`request` An `HttpRequest` instance.

`template` The full name of a template, or a sequence of template names. `Template` instances can also be used.

`context` A dictionary of values to add to the template context. By default, this is an empty dictionary. `Context` objects are also accepted as `context` values.

`status` The HTTP Status code for the response.

`content_type` An alias for `mimetype`. Historically, this parameter was only called `mimetype`, but since this is actually the value included in the HTTP `Content-Type` header, it can also include the character set encoding, which makes it more than just a MIME type specification. If `mimetype` is specified (not `None`), that value is used. Otherwise, `content_type` is used. If neither is given, `DEFAULT_CONTENT_TYPE` is used.

`current_app` A hint indicating which application contains the current view. See the *namespaced URL resolution strategy* for more information.

6.14.3 The rendering process

Before a `TemplateResponse` instance can be returned to the client, it must be rendered. The rendering process takes the intermediate representation of template and context, and turns it into the final byte stream that can be served to the client.

There are three circumstances under which a TemplateResponse will be rendered:

- When the TemplateResponse instance is explicitly rendered, using the `SimpleTemplateResponse.render()` method.
- When the content of the response is explicitly set by assigning `response.content`.

- After passing through template response middleware, but before passing through response middleware.

A TemplateResponse can only be rendered once. The first call to `SimpleTemplateResponse.render()` sets the content of the response; subsequent rendering calls do not change the response content.

However, when `response.content` is explicitly assigned, the change is always applied. If you want to force the content to be re-rendered, you can re-evaluate the rendered content, and assign the content of the response manually:

```
# Set up a rendered TemplateResponse
>>> t = TemplateResponse(request, 'original.html', {})
>>> t.render()
>>> print(t.content)
Original content

# Re-rendering doesn't change content
>>> t.template_name = 'new.html'
>>> t.render()
>>> print(t.content)
Original content

# Assigning content does change, no render() call required
>>> t.content = t.rendered_content
>>> print(t.content)
New content
```

Post-render callbacks

Some operations – such as caching – cannot be performed on an unrendered template. They must be performed on a fully complete and rendered response.

If you're using middleware, the solution is easy. Middleware provides multiple opportunities to process a response on exit from a view. If you put behavior in the Response middleware is guaranteed to execute after template rendering has taken place.

However, if you're using a decorator, the same opportunities do not exist. Any behavior defined in a decorator is handled immediately.

To compensate for this (and any other analogous use cases), `TemplateResponse` allows you to register callbacks that will be invoked when rendering has completed. Using this callback, you can defer critical processing until a point where you can guarantee that rendered content will be available.

To define a post-render callback, just define a function that takes a single argument – response – and register that function with the template response:

```
def my_render_callback(response):
    # Do content-sensitive processing
    do_post_processing()

def my_view(request):
    # Create a response
    response = TemplateResponse(request, 'mytemplate.html', {})
    # Register the callback
    response.add_post_render_callback(my_render_callback)
    # Return the response
    return response
```

`my_render_callback()` will be invoked after the `mytemplate.html` has been rendered, and will be provided the fully rendered `TemplateResponse` instance as an argument.

If the template has already been rendered, the callback will be invoked immediately.

6.14.4 Using TemplateResponse and SimpleTemplateResponse

A TemplateResponse object can be used anywhere that a normal HttpResponse can be used. It can also be used as an alternative to calling `render_to_response()`.

For example, the following simple view returns a `TemplateResponse()` with a simple template, and a context containing a queryset:

```
from django.template.response import TemplateResponse

def blog_index(request):
    return TemplateResponse(request, 'entry_list.html', {'entries': Entry.objects.all()})
```

6.15 Settings

- Available settings
- Deprecated settings

Warning: Be careful when you override settings, especially when the default value is a non-empty tuple or dictionary, such as `MIDDLEWARE_CLASSES` and `TEMPLATE_CONTEXT_PROCESSORS`. Make sure you keep the components required by the features of Django you wish to use.

6.15.1 Available settings

Here's a full list of all available settings, in alphabetical order, and their default values.

ABSOLUTE_URL_OVERRIDES

Default: `{}` (Empty dictionary)

A dictionary mapping `"app_label.model_name"` strings to functions that take a model object and return its URL. This is a way of overriding `get_absolute_url()` methods on a per-installation basis. Example:

```
ABSOLUTE_URL_OVERRIDES = {
    'blogs.weblog': lambda o: "/blogs/%s/" % o.slug,
    'news.story': lambda o: "/stories/%s/%s/" % (o.pub_year, o.slug),
}
```

Note that the model name used in this setting should be all lower-case, regardless of the case of the actual model class name.

ADMIN_FOR

Default: `()` (Empty tuple)

Used for admin-site settings modules, this should be a tuple of settings modules (in the format `'foo.bar.baz'`) for which this site is an admin.

The admin site uses this in its automatically-introspected documentation of models, views and template tags.

ADMINS

Default: `()` (Empty tuple)

A tuple that lists people who get code error notifications. When `DEBUG=False` and a view raises an exception, Django will email these people with the full exception information. Each member of the tuple should be a tuple of (Full name, email address). Example:

```
(('John', 'john@example.com'), ('Mary', 'mary@example.com'))
```

Note that Django will email *all* of these people whenever an error happens. See *Error reporting* for more information.

ALLOWED_INCLUDE_ROOTS

Default: `()` (Empty tuple)

A tuple of strings representing allowed prefixes for the `{% ssi %}` template tag. This is a security measure, so that template authors can't access files that they shouldn't be accessing.

For example, if `ALLOWED_INCLUDE_ROOTS` is `('/home/html', '/var/www')`, then `{% ssi /home/html/foo.txt %}` would work, but `{% ssi /etc/passwd %}` wouldn't.

APPEND_SLASH

Default: `True`

When set to `True`, if the request URL does not match any of the patterns in the URLconf and it doesn't end in a slash, an HTTP redirect is issued to the same URL with a slash appended. Note that the redirect may cause any data submitted in a POST request to be lost.

The `APPEND_SLASH` setting is only used if `CommonMiddleware` is installed (see *Middleware*). See also `PREPEND_WWW`.

AUTHENTICATION_BACKENDS

Default: `('django.contrib.auth.backends.ModelBackend',)`

A tuple of authentication backend classes (as strings) to use when attempting to authenticate a user. See the *authentication backends documentation* for details.

AUTH_PROFILE_MODULE

Default: Not defined

The site-specific user profile model used by this site. See *Storing additional information about users*.

CACHES

New in version 1.3: *Please see the release notes* Default:

```
{
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
    }
}
```

A dictionary containing the settings for all caches to be used with Django. It is a nested dictionary whose contents maps cache aliases to a dictionary containing the options for an individual cache.

The `CACHES` setting must configure a `default` cache; any number of additional caches may also be specified. If you are using a cache backend other than the local memory cache, or you need to define multiple caches, other options will be required. The following cache options are available.

BACKEND

Default: `''` (Empty string)

The cache backend to use. The built-in cache backends are:

- `'django.core.cache.backends.db.DatabaseCache'`
- `'django.core.cache.backends.dummy.DummyCache'`
- `'django.core.cache.backends.filebased.FileBasedCache'`
- `'django.core.cache.backends.locmem.LocMemCache'`
- `'django.core.cache.backends.memcached.MemcachedCache'`
- `'django.core.cache.backends.memcached.PyLibMCCache'`

You can use a cache backend that doesn't ship with Django by setting `BACKEND` to a fully-qualified path of a cache backend class (i.e. `mypackage.backends.whatever.WhateverCache`). Writing a whole new cache backend from scratch is left as an exercise to the reader; see the other backends for examples.

Note: Prior to Django 1.3, you could use a URI based version of the backend name to reference the built-in cache backends (e.g., you could use `'db://tablename'` to refer to the database backend). This format has been deprecated, and will be removed in Django 1.5.

KEY_FUNCTION

A string containing a dotted path to a function that defines how to compose a prefix, version and key into a final cache key. The default implementation is equivalent to the function:

```
def make_key(key, key_prefix, version):
    return ':'.join([key_prefix, str(version), smart_str(key)])
```

You may use any key function you want, as long as it has the same argument signature.

See the *cache documentation* for more information.

KEY_PREFIX

Default: `''` (Empty string)

A string that will be automatically included (prepended by default) to all cache keys used by the Django server.

See the *cache documentation* for more information.

LOCATION

Default: '' (Empty string)

The location of the cache to use. This might be the directory for a file system cache, a host and port for a memcache server, or simply an identifying name for a local memory cache. e.g.:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

OPTIONS

Default: None

Extra parameters to pass to the cache backend. Available parameters vary depending on your cache backend.

Some information on available parameters can be found in the *Cache Backends* documentation. For more information, consult your backend module's own documentation.

TIMEOUT

Default: 300

The number of seconds before a cache entry is considered stale.

VERSION

Default: `1`

The default version number for cache keys generated by the Django server.

See the *cache documentation* for more information.

CACHE_MIDDLEWARE_ALIAS

Default: `default`

The cache connection to use for the cache middleware.

CACHE_MIDDLEWARE_ANONYMOUS_ONLY

Default: `False`

If the value of this setting is `True`, only anonymous requests (i.e., not those made by a logged-in user) will be cached. Otherwise, the middleware caches every page that doesn't have GET or POST parameters.

If you set the value of this setting to `True`, you should make sure you've activated `AuthenticationMiddleware`.

See *Django's cache framework*.

CACHE_MIDDLEWARE_KEY_PREFIX

Default: `''` (Empty string)

The cache key prefix that the cache middleware should use.

See *Django's cache framework*.

CACHE_MIDDLEWARE_SECONDS

Default: `600`

The default number of seconds to cache a page when the caching middleware or `cache_page()` decorator is used.

See *Django's cache framework*.

CSRF_COOKIE_DOMAIN

Default: `None`

The domain to be used when setting the CSRF cookie. This can be useful for easily allowing cross-subdomain requests to be excluded from the normal cross site request forgery protection. It should be set to a string such as `".example.com"` to allow a POST request from a form on one subdomain to be accepted by accepted by a view served from another subdomain.

Please note that the presence of this setting does not imply that Django's CSRF protection is safe from cross-subdomain attacks by default - please see the *CSRF limitations* section.

CSRF_COOKIE_NAME

Default: `'csrftoken'`

The name of the cookie to use for the CSRF authentication token. This can be whatever you want. See *Cross Site Request Forgery protection*.

CSRF_COOKIE_PATH

New in version 1.4: *Please see the release notes* Default: `'/'`

The path set on the CSRF cookie. This should either match the URL path of your Django installation or be a parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths, and each instance will only see its own CSRF cookie.

CSRF_COOKIE_SECURE

New in version 1.4: *Please see the release notes* Default: `False`

Whether to use a secure cookie for the CSRF cookie. If this is set to `True`, the cookie will be marked as "secure," which means browsers may ensure that the cookie is only sent under an HTTPS connection.

CSRF_FAILURE_VIEW

Default: `'django.views.csrf.csrf_failure'`

A dotted path to the view function to be used when an incoming request is rejected by the CSRF protection. The function should have this signature:

```
def csrf_failure(request, reason="")
```

where `reason` is a short message (intended for developers or logging, not for end users) indicating the reason the request was rejected. See *Cross Site Request Forgery protection*.

DATABASES

Default: `{}` (Empty dictionary)

A dictionary containing the settings for all databases to be used with Django. It is a nested dictionary whose contents maps database aliases to a dictionary containing the options for an individual database.

The `DATABASES` setting must configure a `default` database; any number of additional databases may also be specified.

The simplest possible settings file is for a single-database setup using SQLite. This can be configured using the following:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mydatabase'
    }
}
```

For other database backends, or more complex SQLite configurations, other options will be required. The following inner options are available.

ENGINE

Default: `''` (Empty string)

The database backend to use. The built-in database backends are:

- `'django.db.backends.postgresql_psycopg2'`
- `'django.db.backends.mysql'`
- `'django.db.backends.sqlite3'`
- `'django.db.backends.oracle'`

You can use a database backend that doesn't ship with Django by setting `ENGINE` to a fully-qualified path (i.e. `mypackage.backends.whatever`). Writing a whole new database backend from scratch is left as an exercise to the reader; see the other backends for examples.

HOST

Default: `''` (Empty string)

Which host to use when connecting to the database. An empty string means localhost. Not used with SQLite.

If this value starts with a forward slash (`'/'`) and you're using MySQL, MySQL will connect via a Unix socket to the specified socket. For example:

```
"HOST": '/var/run/mysql'
```

If you're using MySQL and this value *doesn't* start with a forward slash, then this value is assumed to be the host.

If you're using PostgreSQL, an empty string means to use a Unix domain socket for the connection, rather than a network connection to localhost. If you explicitly need to use a TCP/IP connection on the local machine with PostgreSQL, specify `localhost` here.

NAME

Default: `''` (Empty string)

The name of the database to use. For SQLite, it's the full path to the database file. When specifying the path, always use forward slashes, even on Windows (e.g. `C:/homes/user/mysite/sqlite3.db`).

OPTIONS

Default: `{}` (Empty dictionary)

Extra parameters to use when connecting to the database. Available parameters vary depending on your database backend.

Some information on available parameters can be found in the *Database Backends* documentation. For more information, consult your backend module's own documentation.

PASSWORD

Default: `''` (Empty string)

The password to use when connecting to the database. Not used with SQLite.

PORT

Default: `''` (Empty string)

The port to use when connecting to the database. An empty string means the default port. Not used with SQLite.

USER

Default: `''` (Empty string)

The username to use when connecting to the database. Not used with SQLite.

TEST_CHARSET

Default: `None`

The character set encoding used to create the test database. The value of this string is passed directly through to the database, so its format is backend-specific.

Supported for the PostgreSQL (`postgresql_psycopg2`) and MySQL (`mysql`) backends.

TEST_COLLATION

Default: `None`

The collation order to use when creating the test database. This value is passed directly to the backend, so its format is backend-specific.

Only supported for the `mysql` backend (see the MySQL manual for details).

TEST_DEPENDENCIES

New in version 1.3: *Please see the release notes* Default: `['default']`, for all databases other than `default`, which has no dependencies.

The creation-order dependencies of the database. See the documentation on *controlling the creation order of test databases* for details.

TEST_MIRROR

Default: `None`

The alias of the database that this database should mirror during testing.

This setting exists to allow for testing of master/slave configurations of multiple databases. See the documentation on *testing master/slave configurations* for details.

TEST_NAME

Default: `None`

The name of database to use when running the test suite.

If the default value (`None`) is used with the SQLite database engine, the tests will use a memory resident database. For all other database engines the test database will use the name `'test_' + DATABASE_NAME`.

See *Testing Django applications*.

TEST_CREATE

Default: `True`

This is an Oracle-specific setting.

If it is set to `False`, the test tablespaces won't be automatically created at the beginning of the tests and dropped at the end.

TEST_USER

Default: `None`

This is an Oracle-specific setting.

The username to use when connecting to the Oracle database that will be used when running tests. If not provided, Django will use `'test_' + USER`.

TEST_USER_CREATE

Default: `True`

This is an Oracle-specific setting.

If it is set to `False`, the test user won't be automatically created at the beginning of the tests and dropped at the end.

TEST_PASSWD

Default: `None`

This is an Oracle-specific setting.

The password to use when connecting to the Oracle database that will be used when running tests. If not provided, Django will use a hardcoded default value.

TEST_TBLSPACE

Default: `None`

This is an Oracle-specific setting.

The name of the tablespace that will be used when running tests. If not provided, Django will use `'test_' + NAME`.

TEST_TBLSPACE_TMP

Default: `None`

This is an Oracle-specific setting.

The name of the temporary tablespace that will be used when running tests. If not provided, Django will use `'test_' + NAME + '_temp'`.

DATABASE_ROUTERS

Default: `[]` (Empty list)

The list of routers that will be used to determine which database to use when performing a database queries.

See the documentation on *automatic database routing in multi database configurations*.

DATE_FORMAT

Default: `'N j, Y'` (e.g. `Feb. 4, 2003`)

The default formatting to use for displaying date fields in any part of the system. Note that if `USE_L10N` is set to `True`, then the locale-dictated format has higher precedence and will be applied instead. See `allowed date format strings`.

See also `DATETIME_FORMAT`, `TIME_FORMAT` and `SHORT_DATE_FORMAT`.

DATE_INPUT_FORMATS

Default:

```
('%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', '%b %d %Y',
'%b %d, %Y', '%d %b %Y', '%d %b, %Y', '%B %d %Y',
'%B %d, %Y', '%d %B %Y', '%d %B, %Y')
```

A tuple of formats that will be accepted when inputting data on a date field. Formats will be tried in order, using the first valid one. Note that these format strings use Python's datetime module syntax, not the format strings from the `date` Django template tag.

When `USE_L10N` is `True`, the locale-dictated format has higher precedence and will be applied instead.

See also `DATETIME_INPUT_FORMATS` and `TIME_INPUT_FORMATS`.

DATETIME_FORMAT

Default: `'N j, Y, P'` (e.g. `Feb. 4, 2003, 4 p.m.`)

The default formatting to use for displaying datetime fields in any part of the system. Note that if `USE_L10N` is set to `True`, then the locale-dictated format has higher precedence and will be applied instead. See `allowed date format strings`.

See also `DATE_FORMAT`, `TIME_FORMAT` and `SHORT_DATETIME_FORMAT`.

DATETIME_INPUT_FORMATS

Default:

```
('%Y-%m-%d %H:%M:%S', '%Y-%m-%d %H:%M', '%Y-%m-%d',
'%m/%d/%Y %H:%M:%S', '%m/%d/%Y %H:%M', '%m/%d/%Y',
'%m/%d/%y %H:%M:%S', '%m/%d/%y %H:%M', '%m/%d/%y')
```

A tuple of formats that will be accepted when inputting data on a datetime field. Formats will be tried in order, using the first valid one. Note that these format strings use Python's datetime module syntax, not the format strings from the `date` Django template tag.

When `USE_L10N` is `True`, the locale-dictated format has higher precedence and will be applied instead.

See also `DATE_INPUT_FORMATS` and `TIME_INPUT_FORMATS`.

DEBUG

Default: `False`

A boolean that turns on/off debug mode.

Never deploy a site into production with `DEBUG` turned on.

Did you catch that? NEVER deploy a site into production with `DEBUG` turned on.

One of the main features of debug mode is the display of detailed error pages. If your app raises an exception when `DEBUG` is `True`, Django will display a detailed traceback, including a lot of metadata about your environment, such as all the currently defined Django settings (from `settings.py`).

As a security measure, Django will *not* include settings that might be sensitive (or offensive), such as `SECRET_KEY` or `PROFANITIES_LIST`. Specifically, it will exclude any setting whose name includes any of the following:

- API
- KEY
- PASS
- PROFANITIES_LIST
- SECRET
- SIGNATURE
- TOKEN

Changed in version 1.4: *Please see the release notes* Note that these are *partial* matches. `'PASS'` will also match PASSWORD, just as `'TOKEN'` will also match TOKENIZED and so on.

Still, note that there are always going to be sections of your debug output that are inappropriate for public consumption. File paths, configuration options and the like all give attackers extra information about your server.

It is also important to remember that when running with `DEBUG` turned on, Django will remember every SQL query it executes. This is useful when you're debugging, but it'll rapidly consume memory on a production server.

DEBUG_PROPAGATE_EXCEPTIONS

Default: `False`

If set to True, Django's normal exception handling of view functions will be suppressed, and exceptions will propagate upwards. This can be useful for some test setups, and should never be used on a live site.

DECIMAL_SEPARATOR

Default: `'.'` (Dot)

Default decimal separator used when formatting decimal numbers.

Note that if `USE_L10N` is set to `True`, then the locale-dictated format has higher precedence and will be applied instead.

See also `NUMBER_GROUPING`, `THOUSAND_SEPARATOR` and `USE_THOUSAND_SEPARATOR`.

DEFAULT_CHARSET

Default: `'utf-8'`

Default charset to use for all `HttpResponse` objects, if a MIME type isn't manually specified. Used with `DEFAULT_CONTENT_TYPE` to construct the `Content-Type` header.

DEFAULT_CONTENT_TYPE

Default: `'text/html'`

Default content type to use for all `HttpResponse` objects, if a MIME type isn't manually specified. Used with `DEFAULT_CHARSET` to construct the `Content-Type` header.

DEFAULT_EXCEPTION_REPORTER_FILTER

Default: `django.views.debug.SafeExceptionReporterFilter`

Default exception reporter filter class to be used if none has been assigned to the `HttpRequest` instance yet. See *Filtering error reports*.

DEFAULT_FILE_STORAGE

Default: `django.core.files.storage.FileSystemStorage`

Default file storage class to be used for any file-related operations that don't specify a particular storage system. See *Managing files*.

DEFAULT_FROM_EMAIL

Default: `'webmaster@localhost'`

Default email address to use for various automated correspondence from the site manager(s).

DEFAULT_INDEX_TABLESPACE

Default: `''` (Empty string)

Default tablespace to use for indexes on fields that don't specify one, if the backend supports it (see *Tablespaces*).

DEFAULT_TABLESPACE

Default: `''` (Empty string)

Default tablespace to use for models that don't specify one, if the backend supports it (see *Tablespaces*).

DISALLOWED_USER_AGENTS

Default: `()` (Empty tuple)

List of compiled regular expression objects representing User-Agent strings that are not allowed to visit any page, systemwide. Use this for bad robots/crawlers. This is only used if `CommonMiddleware` is installed (see *Middleware*).

EMAIL_BACKEND

Default: `'django.core.mail.backends.smtp.EmailBackend'`

The backend to use for sending emails. For the list of available backends see *Sending email*.

EMAIL_FILE_PATH

Default: Not defined

The directory used by the `file` email backend to store output files.

EMAIL_HOST

Default: `'localhost'`

The host to use for sending email.

See also `EMAIL_PORT`.

EMAIL_HOST_PASSWORD

Default: `''` (Empty string)

Password to use for the SMTP server defined in `EMAIL_HOST`. This setting is used in conjunction with `EMAIL_HOST_USER` when authenticating to the SMTP server. If either of these settings is empty, Django won't attempt authentication.

See also `EMAIL_HOST_USER`.

EMAIL_HOST_USER

Default: `''` (Empty string)

Username to use for the SMTP server defined in `EMAIL_HOST`. If empty, Django won't attempt authentication.

See also `EMAIL_HOST_PASSWORD`.

EMAIL_PORT

Default: `25`

Port to use for the SMTP server defined in `EMAIL_HOST`.

EMAIL_SUBJECT_PREFIX

Default: `'[Django] '`

Subject-line prefix for email messages sent with `django.core.mail.mail_admins` or `django.core.mail.mail_managers`. You'll probably want to include the trailing space.

EMAIL_USE_TLS

Default: `False`

Whether to use a TLS (secure) connection when talking to the SMTP server.

FILE_CHARSET

Default: `'utf-8'`

The character encoding used to decode any files read from disk. This includes template files and initial SQL data files.

FILE_UPLOAD_HANDLERS

Default:

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",
 "django.core.files.uploadhandler.TemporaryFileUploadHandler",)
```

A tuple of handlers to use for uploading. See *Managing files* for details.

FILE_UPLOAD_MAX_MEMORY_SIZE

Default: `2621440` (i.e. 2.5 MB).

The maximum size (in bytes) that an upload will be before it gets streamed to the file system. See *Managing files* for details.

FILE_UPLOAD_PERMISSIONS

Default: `None`

The numeric mode (i.e. `0644`) to set newly uploaded files to. For more information about what these modes mean, see the documentation for `os.chmod()`.

If this isn't given or is `None`, you'll get operating-system dependent behavior. On most platforms, temporary files will have a mode of `0600`, and files saved from memory will be saved using the system's standard umask.

Warning: Always prefix the mode with a 0.
If you're not familiar with file modes, please note that the leading `0` is very important: it indicates an octal number, which is the way that modes must be specified. If you try to use `644`, you'll get totally incorrect behavior.

FILE_UPLOAD_TEMP_DIR

Default: `None`

The directory to store data temporarily while uploading files. If `None`, Django will use the standard temporary directory for the operating system. For example, this will default to '/tmp' on *nix-style operating systems.

See *Managing files* for details.

FIRST_DAY_OF_WEEK

Default: `0` (Sunday)

Number representing the first day of the week. This is especially useful when displaying a calendar. This value is only used when not using format internationalization, or when a format cannot be found for the current locale.

The value must be an integer from 0 to 6, where 0 means Sunday, 1 means Monday and so on.

FIXTURE_DIRS

Default: `()` (Empty tuple)

List of directories searched for fixture files, in addition to the `fixtures` directory of each application, in search order.

Note that these paths should use Unix-style forward slashes, even on Windows.

See *Providing initial data with fixtures* and *Fixture loading*.

FORCE_SCRIPT_NAME

Default: `None`

If not `None`, this will be used as the value of the `SCRIPT_NAME` environment variable in any HTTP request. This setting can be used to override the server-provided value of `SCRIPT_NAME`, which may be a rewritten version of the preferred value or not supplied at all.

FORMAT_MODULE_PATH

Default: `None`

A full Python path to a Python package that contains format definitions for project locales. If not `None`, Django will check for a `formats.py` file, under the directory named as the current locale, and will use the formats defined on this file.

For example, if `FORMAT_MODULE_PATH` is set to `mysite.formats`, and current language is `en` (English), Django will expect a directory tree like:

```
mysite/
    formats/
        __init__.py
        en/
            __init__.py
            formats.py
```

Available formats are `DATE_FORMAT`, `TIME_FORMAT`, `DATETIME_FORMAT`, `YEAR_MONTH_FORMAT`, `MONTH_DAY_FORMAT`, `SHORT_DATE_FORMAT`, `SHORT_DATETIME_FORMAT`, `FIRST_DAY_OF_WEEK`, `DECIMAL_SEPARATOR`, `THOUSAND_SEPARATOR` and `NUMBER_GROUPING`.

IGNORABLE_404_URLS

New in version 1.4: *Please see the release notes* Default: `()`

List of compiled regular expression objects describing URLs that should be ignored when reporting HTTP 404 errors via email (see *Error reporting*). Use this if your site does not provide a commonly requested file such as `favicon.ico` or `robots.txt`, or if it gets hammered by script kiddies.

This is only used if `SEND_BROKEN_LINK_EMAILS` is set to `True` and `CommonMiddleware` is installed (see *Middleware*).

INSTALLED_APPS

Default: `()` (Empty tuple)

A tuple of strings designating all applications that are enabled in this Django installation. Each string should be a full Python path to a Python package that contains a Django application, as created by `django-admin.py startapp`.

App names must be unique

The application names (that is, the final dotted part of the path to the module containing `models.py`) defined in `INSTALLED_APPS` *must* be unique. For example, you can't include both `django.contrib.auth` and `myproject.auth` in INSTALLED_APPS.

INTERNAL_IPS

Default: `()` (Empty tuple)

A tuple of IP addresses, as strings, that:

- See debug comments, when `DEBUG` is `True`
- Receive X headers if the `XViewMiddleware` is installed (see *Middleware*)

LANGUAGE_CODE

Default: `'en-us'`

A string representing the language code for this installation. This should be in standard *language format*. For example, U.S. English is `"en-us"`. See *Internationalization and localization*.

LANGUAGE_COOKIE_NAME

Default: `'django_language'`

The name of the cookie to use for the language cookie. This can be whatever you want (but should be different from `SESSION_COOKIE_NAME`). See *Internationalization and localization*.

LANGUAGES

Default: A tuple of all available languages. This list is continually growing and including a copy here would inevitably become rapidly out of date. You can see the current list of translated languages by looking in `django/conf/global_settings.py` (or view the online source).

The list is a tuple of two-tuples in the format `(language code, language name)`, the `language code` part should be a *language name* – for example, `('ja', 'Japanese')`. This specifies which languages are available for language selection. See *Internationalization and localization*.

Generally, the default value should suffice. Only set this setting if you want to restrict language selection to a subset of the Django-provided languages.

If you define a custom `LANGUAGES` setting, it's OK to mark the languages as translation strings (as in the default value referred to above) – but use a "dummy" `gettext()` function, not the one in `django.utils.translation`. You should *never* import `django.utils.translation` from within your settings file, because that module in itself depends on the settings, and that would cause a circular import.

The solution is to use a "dummy" `gettext()` function. Here's a sample settings file:

```
gettext = lambda s: s

LANGUAGES = (
    ('de', gettext('German')),
    ('en', gettext('English')),
)
```

With this arrangement, `django-admin.py makemessages` will still find and mark these strings for translation, but the translation won't happen at runtime – so you'll have to remember to wrap the languages in the *real* `gettext()` in any code that uses `LANGUAGES` at runtime.

LOCALE_PATHS

Default: `()` (Empty tuple)

A tuple of directories where Django looks for translation files. See *How Django discovers translations*.

Example:

```
LOCALE_PATHS = (
    '/home/www/project/common_files/locale',
    '/var/local/translations/locale'
)
```

Note that in the paths you add to the value of this setting, if you have the typical `/path/to/locale/xx/LC_MESSAGES` hierarchy, you should use the path to the `locale` directory (i.e. `'/path/to/locale'`).

LOGGING

New in version 1.3: *Please see the release notes* Default: A logging configuration dictionary.

A data structure containing configuration information. The contents of this data structure will be passed as the argument to the configuration method described in `LOGGING_CONFIG`.

The default logging configuration passes HTTP 500 server errors to an email log handler; all other log messages are given to a NullHandler.

LOGGING_CONFIG

New in version 1.3: *Please see the release notes* Default: `'django.utils.log.dictConfig'`

A path to a callable that will be used to configure logging in the Django project. Points at a instance of Python's dictConfig configuration method by default.

If you set `LOGGING_CONFIG` to `None`, the logging configuration process will be skipped.

LOGIN_REDIRECT_URL

Default: `'/accounts/profile/'`

The URL where requests are redirected after login when the `contrib.auth.login` view gets no `next` parameter.

This is used by the `login_required()` decorator, for example.

Note: You can use `reverse_lazy()` to reference URLs by their name instead of providing a hardcoded value. Assuming a `urls.py` with an URLpattern named `home`:

```
urlpatterns = patterns('',
    url('^welcome/$', 'test_app.views.home', name='home'),
)
```

You can use `reverse_lazy()` like this:

```
from django.core.urlresolvers import reverse_lazy

LOGIN_REDIRECT_URL = reverse_lazy('home')
```

This also works fine with localized URLs using `i18n_patterns()`.

LOGIN_URL

Default: `'/accounts/login/'`

The URL where requests are redirected for login, especially when using the `login_required()` decorator.

Note: See the note on LOGIN_REDIRECT_URL setting

LOGOUT_URL

Default: `'/accounts/logout/'`

LOGIN_URL counterpart.

Note: See the note on LOGIN_REDIRECT_URL setting

MANAGERS

Default: `()` (Empty tuple)

A tuple in the same format as `ADMINS` that specifies who should get broken-link notifications when `SEND_BROKEN_LINK_EMAILS=True`.

MEDIA_ROOT

Default: `''` (Empty string)

Absolute path to the directory that holds media for this installation, used for *managing stored files*.

Example: `"/var/www/example.com/media/"`

See also `MEDIA_URL`.

MEDIA_URL

Default: `''` (Empty string)

URL that handles the media served from `MEDIA_ROOT`, used for *managing stored files*.

Example: `"http://media.example.com/"` Changed in version 1.3: It must end in a slash if set to a non-empty value.

MESSAGE_LEVEL

Default: *messages.INFO*

Sets the minimum message level that will be recorded by the messages framework. See the *messages documentation* for more details.

MESSAGE_STORAGE

Default: `'django.contrib.messages.storage.user_messages.LegacyFallbackStorage'`

Controls where Django stores message data. See the *messages documentation* for more details.

MESSAGE_TAGS

Default:

```
{messages.DEBUG: 'debug',
messages.INFO: 'info',
messages.SUCCESS: 'success',
messages.WARNING: 'warning',
messages.ERROR: 'error',}
```

Sets the mapping of message levels to message tags. See the *messages documentation* for more details.

MIDDLEWARE_CLASSES

Default:

```
('django.middleware.common.CommonMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.middleware.csrf.CsrfViewMiddleware',
 'django.contrib.auth.middleware.AuthenticationMiddleware',
 'django.contrib.messages.middleware.MessageMiddleware',)
```

A tuple of middleware classes to use. See *Middleware*.

MONTH_DAY_FORMAT

Default: `'F j'`

The default formatting to use for date fields on Django admin change-list pages – and, possibly, by other parts of the system – in cases when only the month and day are displayed.

For example, when a Django admin change-list page is being filtered by a date drilldown, the header for a given day displays the day and month. Different locales have different formats. For example, U.S. English would say "January 1," whereas Spanish might say "1 Enero."

See `allowed date format strings`. See also `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT` and `YEAR_MONTH_FORMAT`.

NUMBER_GROUPING

Default: `0`

Number of digits grouped together on the integer part of a number.

Common use is to display a thousand separator. If this setting is `0`, then no grouping will be applied to the number. If this setting is greater than `0`, then `THOUSAND_SEPARATOR` will be used as the separator between those groups.

Note that if `USE_L10N` is set to `True`, then the locale-dictated format has higher precedence and will be applied instead.

See also `DECIMAL_SEPARATOR`, `THOUSAND_SEPARATOR` and `USE_THOUSAND_SEPARATOR`.

PASSWORD_RESET_TIMEOUT_DAYS

Default: `3`

The number of days a password reset link is valid for. Used by the `django.contrib.auth` password reset mechanism.

PREPEND_WWW

Default: `False`

Whether to prepend the "www." subdomain to URLs that don't have it. This is only used if `CommonMiddleware` is installed (see *Middleware*). See also `APPEND_SLASH`.

PROFANITIES_LIST

Default: `()` (Empty tuple)

A tuple of profanities, as strings, that will be forbidden in comments when `COMMENTS_ALLOW_PROFANITIES` is `False`.

RESTRUCTUREDTEXT_FILTER_SETTINGS

Default: `{}`

A dictionary containing settings for the `restructuredtext` markup filter from the *django.contrib.markup application*. They override the default writer settings. See the Docutils restructuredtext writer settings docs for details.

ROOT_URLCONF

Default: Not defined

A string representing the full Python import path to your root URLconf. For example: `"mydjangoapps.urls"`. Can be overridden on a per-request basis by setting the attribute `urlconf` on the incoming `HttpRequest` object. See *How Django processes a request* for details.

SECRET_KEY

Default: '' (Empty string)

A secret key for this particular Django installation. Used to provide a seed in secret-key hashing algorithms. Set this to a random string – the longer, the better. `django-admin.py startproject` creates one automatically.

SECURE_PROXY_SSL_HEADER

New in version 1.4: *Please see the release notes* Default: `None`

A tuple representing a HTTP header/value combination that signifies a request is secure. This controls the behavior of the request object's `is_secure()` method.

This takes some explanation. By default, `is_secure()` is able to determine whether a request is secure by looking at whether the requested URL uses "https://". This is important for Django's CSRF protection, and may be used by your own code or third-party apps.

If your Django app is behind a proxy, though, the proxy may be "swallowing" the fact that a request is HTTPS, using a non-HTTPS connection between the proxy and Django. In this case, `is_secure()` would always return `False` – even for requests that were made via HTTPS by the end user.

In this situation, you'll want to configure your proxy to set a custom HTTP header that tells Django whether the request came in via HTTPS, and you'll want to set `SECURE_PROXY_SSL_HEADER` so that Django knows what header to look for.

You'll need to set a tuple with two elements – the name of the header to look for and the required value. For example:

```
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTOCOL', 'https')
```

Here, we're telling Django that we trust the `X-Forwarded-Protocol` header that comes from our proxy, and any time its value is `'https'`, then the request is guaranteed to be secure (i.e., it originally came in via HTTPS). Obviously, you should *only* set this setting if you control your proxy or have some other guarantee that it sets/strips this header appropriately.

Note that the header needs to be in the format as used by `request.META` – all caps and likely starting with `HTTP_`. (Remember, Django automatically adds `'HTTP_'` to the start of x-header names before making the header available in `request.META`.)

Warning: You will probably open security holes in your site if you set this without knowing what you're doing. And if you fail to set it when you should. Seriously.
Make sure ALL of the following are true before setting this (assuming the values from the example above):

- Your Django app is behind a proxy.
- Your proxy strips the 'X-Forwarded-Protocol' header from all incoming requests. In other words, if end users include that header in their requests, the proxy will discard it.
- Your proxy sets the 'X-Forwarded-Protocol' header and sends it to Django, but only for requests that originally come in via HTTPS.

If any of those are not true, you should keep this setting set to `None` and find another way of determining HTTPS, perhaps via custom middleware.

SEND_BROKEN_LINK_EMAILS

Default: `False`

Whether to send an email to the `MANAGERS` each time somebody visits a Django-powered page that is 404ed with a non-empty referer (i.e., a broken link). This is only used if `CommonMiddleware` is installed (see *Middleware*). See also `IGNORABLE_404_URLS` and *Error reporting*.

SERIALIZATION_MODULES

Default: Not defined.

A dictionary of modules containing serializer definitions (provided as strings), keyed by a string identifier for that serialization type. For example, to define a YAML serializer, use:

```
SERIALIZATION_MODULES = { 'yaml' : 'path.to.yaml_serializer' }
```

SERVER_EMAIL

Default: `'root@localhost'`

The email address that error messages come from, such as those sent to `ADMINS` and `MANAGERS`.

SESSION_COOKIE_AGE

Default: `1209600` (2 weeks, in seconds)

The age of session cookies, in seconds. See *How to use sessions*.

SESSION_COOKIE_DOMAIN

Default: `None`

The domain to use for session cookies. Set this to a string such as `".example.com"` for cross-domain cookies, or use `None` for a standard domain cookie. See the *How to use sessions*.

SESSION_COOKIE_HTTPONLY

Default: `True`

Whether to use HTTPOnly flag on the session cookie. If this is set to `True`, client-side JavaScript will not to be able to access the session cookie.

HTTPOnly is a flag included in a Set-Cookie HTTP response header. It is not part of the **RFC 2109** standard for cookies, and it isn't honored consistently by all browsers. However, when it is honored, it can be a useful way to mitigate the risk of client side script accessing the protected cookie data. Changed in version 1.4: The default value of the setting was changed from `False` to `True`.

SESSION_COOKIE_NAME

Default: `'sessionid'`

The name of the cookie to use for sessions. This can be whatever you want (but should be different from `LANGUAGE_COOKIE_NAME`). See the *How to use sessions*.

SESSION_COOKIE_PATH

Default: `'/'`

The path set on the session cookie. This should either match the URL path of your Django installation or be parent of that path.

This is useful if you have multiple Django instances running under the same hostname. They can use different cookie paths, and each instance will only see its own session cookie.

SESSION_COOKIE_SECURE

Default: `False`

Whether to use a secure cookie for the session cookie. If this is set to `True`, the cookie will be marked as "secure," which means browsers may ensure that the cookie is only sent under an HTTPS connection. See the *How to use sessions*.

SESSION_ENGINE

Default: `django.contrib.sessions.backends.db`

Controls where Django stores session data. Valid values are:

- `'django.contrib.sessions.backends.db'`
- `'django.contrib.sessions.backends.file'`
- `'django.contrib.sessions.backends.cache'`
- `'django.contrib.sessions.backends.cached_db'`
- `'django.contrib.sessions.backends.signed_cookies'`

See *How to use sessions*.

SESSION_EXPIRE_AT_BROWSER_CLOSE

Default: `False`

Whether to expire the session when the user closes his or her browser. See the *How to use sessions*.

SESSION_FILE_PATH

Default: `None`

If you're using file-based session storage, this sets the directory in which Django will store session data. See *How to use sessions*. When the default value (`None`) is used, Django will use the standard temporary directory for the system.

SESSION_SAVE_EVERY_REQUEST

Default: `False`

Whether to save the session data on every request. See *How to use sessions*.

SHORT_DATE_FORMAT

Default: `m/d/Y` (e.g. `12/31/2003`)

An available formatting that can be used for displaying date fields on templates. Note that if `USE_L10N` is set to `True`, then the corresponding locale-dictated format has higher precedence and will be applied. See `allowed date format strings`.

See also DATE_FORMAT and SHORT_DATETIME_FORMAT.

SHORT_DATETIME_FORMAT

Default: m/d/Y P (e.g. 12/31/2003 4 p.m.)

An available formatting that can be used for displaying datetime fields on templates. Note that if USE_L10N is set to True, then the corresponding locale-dictated format has higher precedence and will be applied. See allowed date format strings.

See also DATE_FORMAT and SHORT_DATE_FORMAT.

SIGNING_BACKEND

New in version 1.4: *Please see the release notes* Default: 'django.core.signing.TimestampSigner'

The backend used for signing cookies and other data.

See also the *Cryptographic signing* documentation.

SITE_ID

Default: Not defined

The ID, as an integer, of the current site in the django_site database table. This is used so that application data can hook into specific site(s) and a single database can manage content for multiple sites.

See *The "sites" framework*.

STATIC_ROOT

Default: '' (Empty string)

The absolute path to the directory where collectstatic will collect static files for deployment.

Example: "/var/www/example.com/static/"

If the *staticfiles* contrib app is enabled (default) the collectstatic management command will collect static files into this directory. See the howto on *managing static files* for more details about usage.

> **Warning:** This should be an (initially empty) destination directory for collecting your static files from their permanent locations into one directory for ease of deployment; it is **not** a place to store your static files permanently. You should do that in directories that will be found by *staticfiles*'s finders, which by default, are 'static/' app sub-directories and any directories you include in STATICFILES_DIRS).

See *staticfiles reference* and STATIC_URL.

STATIC_URL

Default: None

URL to use when referring to static files located in STATIC_ROOT.

Example: "/static/" or "http://static.example.com/"

If not None, this will be used as the base path for *media definitions* and the *staticfiles app*.

It must end in a slash if set to a non-empty value.

See STATIC_ROOT.

TEMPLATE_CONTEXT_PROCESSORS

Default:

```
("django.contrib.auth.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media",
"django.core.context_processors.static",
"django.core.context_processors.tz",
"django.contrib.messages.context_processors.messages")
```

A tuple of callables that are used to populate the context in RequestContext. These callables take a request object as their argument and return a dictionary of items to be merged into the context. New in version 1.3: The django.core.context_processors.static context processor was added in this release.New in version 1.4: The django.core.context_processors.tz context processor was added in this release.

TEMPLATE_DEBUG

Default: False

A boolean that turns on/off template debug mode. If this is True, the fancy error page will display a detailed report for any exception raised during template rendering. This report contains the relevant snippet of the template, with the appropriate line highlighted.

Note that Django only displays fancy error pages if DEBUG is True, so you'll want to set that to take advantage of this setting.

See also DEBUG.

TEMPLATE_DIRS

Default: () (Empty tuple)

List of locations of the template source files searched by django.template.loaders.filesystem.Loader, in search order.

Note that these paths should use Unix-style forward slashes, even on Windows.

See *The Django template language*.

TEMPLATE_LOADERS

Default:

```
('django.template.loaders.filesystem.Loader',
 'django.template.loaders.app_directories.Loader')
```

A tuple of template loader classes, specified as strings. Each Loader class knows how to import templates from a particular source. Optionally, a tuple can be used instead of a string. The first item in the tuple should be the Loader's module, subsequent items are passed to the Loader during initialization. See *The Django template language: For Python programmers*.

TEMPLATE_STRING_IF_INVALID

Default: '' (Empty string)

Output, as a string, that the template system should use for invalid (e.g. misspelled) variables. See *How invalid variables are handled*..

TEST_RUNNER

Default: `'django.test.simple.DjangoTestSuiteRunner'`

The name of the class to use for starting the test suite. See *Testing Django applications*.

THOUSAND_SEPARATOR

Default: `,` (Comma)

Default thousand separator used when formatting numbers. This setting is used only when `USE_THOUSAND_SEPARATOR` is `True` and `NUMBER_GROUPING` is greater than `0`.

Note that if `USE_L10N` is set to `True`, then the locale-dictated format has higher precedence and will be applied instead.

See also `NUMBER_GROUPING`, `DECIMAL_SEPARATOR` and `USE_THOUSAND_SEPARATOR`.

TIME_FORMAT

Default: `'P'` (e.g. `4 p.m.`)

The default formatting to use for displaying time fields in any part of the system. Note that if `USE_L10N` is set to `True`, then the locale-dictated format has higher precedence and will be applied instead. See `allowed date format strings`.

See also `DATE_FORMAT` and `DATETIME_FORMAT`.

TIME_INPUT_FORMATS

Default: `('%H:%M:%S', '%H:%M')`

A tuple of formats that will be accepted when inputting data on a time field. Formats will be tried in order, using the first valid one. Note that these format strings use Python's datetime module syntax, not the format strings from the `date` Django template tag.

When `USE_L10N` is `True`, the locale-dictated format has higher precedence and will be applied instead.

See also `DATE_INPUT_FORMATS` and `DATETIME_INPUT_FORMATS`.

TIME_ZONE

Default: `'America/Chicago'` Changed in version 1.4: The meaning of this setting now depends on the value of `USE_TZ`. A string representing the time zone for this installation, or `None`. See available choices. (Note that list of available choices lists more than one on the same line; you'll want to use just one of the choices for a given time zone. For instance, one line says `'Europe/London GB GB-Eire'`, but you should use the first bit of that – `'Europe/London'` – as your `TIME_ZONE` setting.)

Note that this isn't necessarily the time zone of the server. For example, one server may serve multiple Django-powered sites, each with a separate time zone setting.

When `USE_TZ` is `False`, this is the time zone in which Django will store all datetimes. When `USE_TZ` is `True`, this is the default time zone that Django will use to display datetimes in templates and to interpret datetimes entered in forms.

Django sets the `os.environ['TZ']` variable to the time zone you specify in the `TIME_ZONE` setting. Thus, all your views and models will automatically operate in this time zone. However, Django won't set the `TZ` environment variable under the following conditions:

- If you're using the manual configuration option as described in *manually configuring settings*, or
- If you specify `TIME_ZONE = None`. This will cause Django to fall back to using the system timezone. However, this is discouraged when `USE_TZ = True`, because it makes conversions between local time and UTC less reliable.

If Django doesn't set the `TZ` environment variable, it's up to you to ensure your processes are running in the correct environment.

Note: Django cannot reliably use alternate time zones in a Windows environment. If you're running Django on Windows, `TIME_ZONE` must be set to match the system time zone.

USE_ETAGS

Default: `False`

A boolean that specifies whether to output the "Etag" header. This saves bandwidth but slows down performance. This is used by the `CommonMiddleware` (see *Middleware*) and in the''Cache Framework'' (see *Django's cache framework*).

USE_I18N

Default: `True`

A boolean that specifies whether Django's translation system should be enabled. This provides an easy way to turn it off, for performance. If this is set to `False`, Django will make some optimizations so as not to load the translation machinery.

See also `LANGUAGE_CODE`, `USE_L10N` and `USE_TZ`.

USE_L10N

Default: `False`

A boolean that specifies if localized formatting of data will be enabled by default or not. If this is set to `True`, e.g. Django will display numbers and dates using the format of the current locale.

See also `LANGUAGE_CODE`, `USE_I18N` and `USE_TZ`.

Note: The default `settings.py` file created by `django-admin.py startproject` includes `USE_L10N = True` for convenience.

USE_THOUSAND_SEPARATOR

Default: `False`

A boolean that specifies whether to display numbers using a thousand separator. When `USE_L10N` is set to `True` and if this is also set to `True`, Django will use the values of `THOUSAND_SEPARATOR` and `NUMBER_GROUPING` to format numbers.

See also `DECIMAL_SEPARATOR`, `NUMBER_GROUPING` and `THOUSAND_SEPARATOR`.

USE_TZ

New in version 1.4: *Please see the release notes* Default: `False`

A boolean that specifies if datetimes will be timezone-aware by default or not. If this is set to `True`, Django will use timezone-aware datetimes internally. Otherwise, Django will use naive datetimes in local time.

See also `TIME_ZONE`, `USE_I18N` and `USE_L10N`.

Note: The default `settings.py` file created by `django-admin.py startproject` includes `USE_TZ = True` for convenience.

USE_X_FORWARDED_HOST

New in version 1.3.1: *Please see the release notes* Default: `False`

A boolean that specifies whether to use the X-Forwarded-Host header in preference to the Host header. This should only be enabled if a proxy which sets this header is in use.

WSGI_APPLICATION

New in version 1.4: *Please see the release notes* Default: `None`

The full Python path of the WSGI application object that Django's built-in servers (e.g. `runserver`) will use. The `django-admin.py startproject` management command will create a simple `wsgi.py` file with an `application` callable in it, and point this setting to that `application`.

If not set, the return value of `django.core.wsgi.get_wsgi_application()` will be used. In this case, the behavior of `runserver` will be identical to previous Django versions.

YEAR_MONTH_FORMAT

Default: `'F Y'`

The default formatting to use for date fields on Django admin change-list pages – and, possibly, by other parts of the system – in cases when only the year and month are displayed.

For example, when a Django admin change-list page is being filtered by a date drilldown, the header for a given month displays the month and the year. Different locales have different formats. For example, U.S. English would say "January 2006," whereas another locale might say "2006/January."

See `allowed date format strings`. See also `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT` and `MONTH_DAY_FORMAT`.

X_FRAME_OPTIONS

Default: `'SAMEORIGIN'`

The default value for the X-Frame-Options header used by `XFrameOptionsMiddleware`. See the *clickjacking protection* documentation.

6.15.2 Deprecated settings

ADMIN_MEDIA_PREFIX

Deprecated since version 1.4: This setting has been obsoleted by the `django.contrib.staticfiles` app integration. See the *Django 1.4 release notes* for more information.

IGNORABLE_404_ENDS

Deprecated since version 1.4: This setting has been superseded by `IGNORABLE_404_URLS`.

IGNORABLE_404_STARTS

Deprecated since version 1.4: This setting has been superseded by `IGNORABLE_404_URLS`.

URL_VALIDATOR_USER_AGENT

Deprecated since version 1.5: This value was used as the `User-Agent` header when checking if a URL exists, a feature that was removed due to security and performance issues.

6.16 Signals

A list of all the signals that Django sends.

See Also:

See the documentation on the *signal dispatcher* for information regarding how to register for and receive signals.

The *comment framework* sends a *set of comment-related signals*.

The *authentication framework* sends *signals when a user is logged in / out*.

6.16.1 Model signals

The `django.db.models.signals` module defines a set of signals sent by the module system.

Warning: Many of these signals are sent by various model methods like `__init__()` or `save()` that you can overwrite in your own code.
If you override these methods on your model, you must call the parent class' methods for this signals to be sent.
Note also that Django stores signal handlers as weak references by default, so if your handler is a local function, it may be garbage collected. To prevent this, pass `weak=False` when you call the signal's `connect()`.

pre_init

`django.db.models.signals.`**`pre_init`**

Whenever you instantiate a Django model,, this signal is sent at the beginning of the model's `__init__()` method.

Arguments sent with this signal:

`sender` The model class that just had an instance created.

`args` A list of positional arguments passed to `__init__()`:

`kwargs` A dictionary of keyword arguments passed to `__init__()`:.

For example, the *tutorial* has this line:

```
p = Poll(question="What's up?", pub_date=datetime.now())
```

The arguments sent to a `pre_init` handler would be:

Argument	Value
`sender`	`Poll` (the class itself)
`args`	`[]` (an empty list because there were no positional arguments passed to `__init__`.)
`kwargs`	`{'question': "What's up?", 'pub_date': datetime.now()}`

post_init

`django.db.models.signals.`**`post_init`**

Like pre_init, but this one is sent when the `__init__()`: method finishes.

Arguments sent with this signal:

`sender` As above: the model class that just had an instance created.

`instance` The actual instance of the model that's just been created.

pre_save

`django.db.models.signals.`**`pre_save`**

This is sent at the beginning of a model's `save()` method.

Arguments sent with this signal:

`sender` The model class.

`instance` The actual instance being saved.

`raw` A boolean; `True` if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

New in version 1.3: *Please see the release notes*

`using` The database alias being used.

New in version 1.5: *Please see the release notes*

`update_fields` The set of fields to update explicitly specified in the `save()` method. `None` if this argument was not used in the `save()` call.

post_save

`django.db.models.signals.`**`post_save`**

Like `pre_save`, but sent at the end of the `save()` method.

Arguments sent with this signal:

`sender` The model class.

`instance` The actual instance being saved.

`created` A boolean; `True` if a new record was created.

`raw` A boolean; `True` if the model is saved exactly as presented (i.e. when loading a fixture). One should not query/modify other records in the database as the database might not be in a consistent state yet.

New in version 1.3: *Please see the release notes*

`using` The database alias being used.

New in version 1.5: *Please see the release notes*

`update_fields` The set of fields to update explicitly specified in the `save()` method. `None` if this argument was not used in the `save()` call.

pre_delete

`django.db.models.signals.`**`pre_delete`**

Sent at the beginning of a model's `delete()` method and a queryset's `delete()` method.

Arguments sent with this signal:

`sender` The model class.

`instance` The actual instance being deleted.

New in version 1.3: *Please see the release notes*

`using` The database alias being used.

post_delete

`django.db.models.signals.`**`post_delete`**

Like `pre_delete`, but sent at the end of a model's `delete()` method and a queryset's `delete()` method.

Arguments sent with this signal:

`sender` The model class.

`instance` The actual instance being deleted.

Note that the object will no longer be in the database, so be very careful what you do with this instance.

New in version 1.3: *Please see the release notes*

`using` The database alias being used.

m2m_changed

`django.db.models.signals.`**`m2m_changed`**

Sent when a `ManyToManyField` is changed on a model instance. Strictly speaking, this is not a model signal since it is sent by the `ManyToManyField`, but since it complements the `pre_save`/`post_save` and `pre_delete`/`post_delete` when it comes to tracking changes to models, it is included here.

Arguments sent with this signal:

`sender` The intermediate model class describing the `ManyToManyField`. This class is automatically created when a many-to-many field is defined; you can access it using the `through` attribute on the many-to-many field.

`instance` The instance whose many-to-many relation is updated. This can be an instance of the `sender`, or of the class the `ManyToManyField` is related to.

`action` A string indicating the type of update that is done on the relation. This can be one of the following:

`"pre_add"` Sent *before* one or more objects are added to the relation.

`"post_add"` Sent *after* one or more objects are added to the relation.

`"pre_remove"` Sent *before* one or more objects are removed from the relation.

`"post_remove"` Sent *after* one or more objects are removed from the relation.

`"pre_clear"` Sent *before* the relation is cleared.

`"post_clear"` Sent *after* the relation is cleared.

`reverse` Indicates which side of the relation is updated (i.e., if it is the forward or reverse relation that is being modified).

`model` The class of the objects that are added to, removed from or cleared from the relation.

`pk_set` For the `pre_add`, `post_add`, `pre_remove` and `post_remove` actions, this is a list of primary key values that have been added to or removed from the relation.

For the `pre_clear` and `post_clear` actions, this is `None`.

New in version 1.3: *Please see the release notes*

`using` The database alias being used.

For example, if a `Pizza` can have multiple `Topping` objects, modeled like this:

```
class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

If we would do something like this:

```
>>> p = Pizza.object.create(...)
>>> t = Topping.objects.create(...)
>>> p.toppings.add(t)
```

the arguments sent to a `m2m_changed` handler would be:

Argument	Value
sender	Pizza.toppings.through (the intermediate m2m class)
instance	p (the Pizza instance being modified)
action	"pre_add" (followed by a separate signal with "post_add")
reverse	False (Pizza contains the ManyToManyField, so this call modifies the forward relation)
model	Topping (the class of the objects added to the Pizza)
pk_set	[t.id] (since only Topping t was added to the relation)
using	"default" (since the default router sends writes here)

And if we would then do something like this:

```
>>> t.pizza_set.remove(p)
```

the arguments sent to a m2m_changed handler would be:

Argument	Value
sender	Pizza.toppings.through (the intermediate m2m class)
instance	t (the Topping instance being modified)
action	"pre_remove" (followed by a separate signal with "post_remove")
reverse	True (Pizza contains the ManyToManyField, so this call modifies the reverse relation)
model	Pizza (the class of the objects removed from the Topping)
pk_set	[p.id] (since only Pizza p was removed from the relation)
using	"default" (since the default router sends writes here)

class_prepared

django.db.models.signals.**class_prepared**

Sent whenever a model class has been "prepared" – that is, once model has been defined and registered with Django's model system. Django uses this signal internally; it's not generally used in third-party applications.

Arguments that are sent with this signal:

sender The model class which was just prepared.

6.16.2 Management signals

Signals sent by *django-admin*.

post_syncdb

django.db.models.signals.**post_syncdb**

Sent by the syncdb command after it installs an application, and the flush command.

Any handlers that listen to this signal need to be written in a particular place: a management module in one of your INSTALLED_APPS. If handlers are registered anywhere else they may not be loaded by syncdb. It is important that handlers of this signal perform idempotent changes (e.g. no database alterations) as this may cause the flush management command to fail if it also ran during the syncdb command.

Arguments sent with this signal:

sender The models module that was just installed. That is, if syncdb just installed an app called "foo.bar.myapp", sender will be the foo.bar.myapp.models module.

app Same as sender.

created_models A list of the model classes from any app which `syncdb` has created so far.

verbosity Indicates how much information manage.py is printing on screen. See the `--verbosity` flag for details.

Functions which listen for `post_syncdb` should adjust what they output to the screen based on the value of this argument.

interactive If `interactive` is `True`, it's safe to prompt the user to input things on the command line. If `interactive` is `False`, functions which listen for this signal should not try to prompt for anything.

For example, the `django.contrib.auth` app only prompts to create a superuser when `interactive` is `True`.

For example, `yourapp/management/__init__.py` could be written like:

```
from django.db.models.signals import post_syncdb
import yourapp.models

def my_callback(sender, **kwargs):
    # Your specific logic here
    pass

post_syncdb.connect(my_callback, sender=yourapp.models)
```

6.16.3 Request/response signals

Signals sent by the core framework when processing a request.

request_started

`django.core.signals.`**request_started**

Sent when Django begins processing an HTTP request.

Arguments sent with this signal:

sender The handler class – e.g. `django.core.handlers.wsgi.WsgiHandler` – that handled the request.

request_finished

`django.core.signals.`**request_finished**

Sent when Django finishes processing an HTTP request.

Arguments sent with this signal:

sender The handler class, as above.

got_request_exception

`django.core.signals.`**got_request_exception**

This signal is sent whenever Django encounters an exception while processing an incoming HTTP request.

Arguments sent with this signal:

sender The handler class, as above.

request The `HttpRequest` object.

6.16.4 Test signals

Signals only sent when *running tests*.

setting_changed

New in version 1.4: *Please see the release notes*

`django.test.signals.`**`setting_changed`**

This signal is sent when the value of a setting is changed through the `django.test.TestCase.setting()` context manager or the `django.test.utils.override_settings()` decorator/context manager.

It's actually sent twice: when the new value is applied ("setup") and when the original value is restored ("teardown").

Arguments sent with this signal:

sender The settings handler.

setting The name of the setting.

value The value of the setting after the change. For settings that initially don't exist, in the "teardown" phase, `value` is `None`.

template_rendered

`django.test.signals.`**`template_rendered`**

Sent when the test system renders a template. This signal is not emitted during normal operation of a Django server – it is only available during testing.

Arguments sent with this signal:

sender The `Template` object which was rendered.

template Same as sender

context The `Context` with which the template was rendered.

6.16.5 Database Wrappers

Signals sent by the database wrapper when a database connection is initiated.

connection_created

`django.db.backends.signals.`**`connection_created`**

Sent when the database wrapper makes the initial connection to the database. This is particularly useful if you'd like to send any post connection commands to the SQL backend.

Arguments sent with this signal:

sender The database wrapper class – i.e. `django.db.backends.postgresql_psycopg2.DatabaseWrapper` or `django.db.backends.mysql.DatabaseWrapper`, etc.

connection The database connection that was opened. This can be used in a multiple-database configuration to differentiate connection signals from different databases.

6.17 Templates

Django's template engine provides a powerful mini-language for defining the user-facing layer of your application, encouraging a clean separation of application and presentation logic. Templates can be maintained by anyone with an understanding of HTML; no knowledge of Python is required.

6.17.1 Built-in template tags and filters

This document describes Django's built-in template tags and filters. It is recommended that you use the *automatic documentation*, if available, as this will also include documentation for any custom tags or filters installed.

Built-in tag reference

autoescape

Controls the current auto-escaping behavior. This tag takes either `on` or `off` as an argument and that determines whether auto-escaping is in effect inside the block. The block is closed with an `endautoescape` ending tag.

When auto-escaping is in effect, all variable content has HTML escaping applied to it before placing the result into the output (but after any filters have been applied). This is equivalent to manually applying the `escape` filter to each variable.

The only exceptions are variables that are already marked as "safe" from escaping, either by the code that populated the variable, or because it has had the `safe` or `escape` filters applied.

Sample usage:

```
{% autoescape on %}
    {{ body }}
{% endautoescape %}
```

block

Defines a block that can be overridden by child templates. See *Template inheritance* for more information.

comment

Ignores everything between `{% comment %}` and `{% endcomment %}`.

csrf_token

This tag is used for CSRF protection, as described in the documentation for *Cross Site Request Forgeries*.

cycle

Cycles among the given strings or variables each time this tag is encountered.

Within a loop, cycles among the given strings each time through the loop:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}">
        ...
    </tr>
{% endfor %}
```

You can use variables, too. For example, if you have two template variables, `rowvalue1` and `rowvalue2`, you can cycle between their values like this:

```
{% for o in some_list %}
    <tr class="{% cycle rowvalue1 rowvalue2 %}">
        ...
    </tr>
{% endfor %}
```

Note that variable arguments (`rowvalue1` and `rowvalue2` above) are NOT auto-escaped! So either make sure that you trust their values, or use explicit escaping, like this:

```
{% for o in some_list %}
    <tr class="{% filter force_escape %}{% cycle rowvalue1 rowvalue2 %}{% endfilter %}">
        ...
    </tr>
{% endfor %}
```

You can mix variables and strings:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' rowvalue2 'row3' %}">
        ...
    </tr>
{% endfor %}
```

In some cases you might want to refer to the next value of a cycle from outside of a loop. To do this, just give the `{% cycle %}` tag a name, using "as", like this:

```
{% cycle 'row1' 'row2' as rowcolors %}
```

From then on, you can insert the current value of the cycle wherever you'd like in your template by referencing the cycle name as a context variable. If you want to move the cycle onto the next value, you use the cycle tag again, using the name of the variable. So, the following template:

```
<tr>
    <td class="{% cycle 'row1' 'row2' as rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
<tr>
    <td class="{% cycle rowcolors %}">...</td>
    <td class="{{ rowcolors }}">...</td>
</tr>
```

would output:

```
<tr>
    <td class="row1">...</td>
```

```
    <td class="row1">...</td>
</tr>
<tr>
    <td class="row2">...</td>
    <td class="row2">...</td>
</tr>
```

You can use any number of values in a `{% cycle %}` tag, separated by spaces. Values enclosed in single (') or double quotes (") are treated as string literals, while values without quotes are treated as template variables.

Note that the variables included in the cycle will not be escaped. This is because template tags do not escape their content. Any HTML or Javascript code contained in the printed variable will be rendered as-is, which could potentially lead to security issues.

For backwards compatibility, the `{% cycle %}` tag supports the much inferior old syntax from previous Django versions. You shouldn't use this in any new projects, but for the sake of the people who are still using it, here's what it looks like:

```
{% cycle row1,row2,row3 %}
```

In this syntax, each value gets interpreted as a literal string, and there's no way to specify variable values. Or literal commas. Or spaces. Did we mention you shouldn't use this syntax in any new projects? New in version 1.3: *Please see the release notes* By default, when you use the `as` keyword with the cycle tag, the usage of `{% cycle %}` that declares the cycle will itself output the first value in the cycle. This could be a problem if you want to use the value in a nested loop or an included template. If you want to just declare the cycle, but not output the first value, you can add a `silent` keyword as the last keyword in the tag. For example:

```
{% for obj in some_list %}
    {% cycle 'row1' 'row2' as rowcolors silent %}
    <tr class="{{ rowcolors }}">{% include "subtemplate.html " %}</tr>
{% endfor %}
```

This will output a list of `<tr>` elements with `class` alternating between `row1` and `row2`; the subtemplate will have access to `rowcolors` in it's context that matches the class of the `<tr>` that encloses it. If the `silent` keyword were to be omitted, `row1` would be emitted as normal text, outside the `<tr>` element.

When the silent keyword is used on a cycle definition, the silence automatically applies to all subsequent uses of the cycle tag. In, the following template would output *nothing*, even though the second call to `{% cycle %}` doesn't specify silent:

```
{% cycle 'row1' 'row2' as rowcolors silent %}
{% cycle rowcolors %}
```

debug

Outputs a whole load of debugging information, including the current context and imported modules.

extends

Signals that this template extends a parent template.

This tag can be used in two ways:

- `{% extends "base.html" %}` (with quotes) uses the literal value `"base.html"` as the name of the parent template to extend.

- `{% extends variable %}` uses the value of `variable`. If the variable evaluates to a string, Django will use that string as the name of the parent template. If the variable evaluates to a `Template` object, Django will use that object as the parent template.

See *Template inheritance* for more information.

filter

Filters the contents of the variable through variable filters.

Filters can also be piped through each other, and they can have arguments – just like in variable syntax.

Sample usage:

```
{% filter force_escape|lower %}
    This text will be HTML-escaped, and will appear in all lowercase.
{% endfilter %}
```

Note: The `escape` and `safe` filters are not acceptable arguments. Instead, use the `autoescape` tag to manage autoescaping for blocks of template code.

---

firstof

Outputs the first variable passed that is not False. Does NOT auto-escape variable values.

Outputs nothing if all the passed variables are False.

Sample usage:

```
{% firstof var1 var2 var3 %}
```

This is equivalent to:

```
{% if var1 %}
    {{ var1|safe }}
{% else %}{% if var2 %}
    {{ var2|safe }}
{% else %}{% if var3 %}
    {{ var3|safe }}
{% endif %}{% endif %}{% endif %}
```

You can also use a literal string as a fallback value in case all passed variables are False:

```
{% firstof var1 var2 var3 "fallback value" %}
```

Note that the variables included in the firstof tag will not be escaped. This is because template tags do not escape their content. Any HTML or Javascript code contained in the printed variable will be rendered as-is, which could potentially lead to security issues. If you need to escape the variables in the firstof tag, you must do so explicitly:

```
{% filter force_escape %}
    {% firstof var1 var2 var3 "fallback value" %}
{% endfilter %}
```

for

Loop over each item in an array. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

You can loop over a list in reverse by using `{% for obj in list reversed %}`.

If you need to loop over a list of lists, you can unpack the values in each sub-list into individual variables. For example, if your context contains a list of (x,y) coordinates called `points`, you could use the following to output the list of points:

```
{% for x, y in points %}
    There is a point at {{ x }},{{ y }}
{% endfor %}
```

This can also be useful if you need to access the items in a dictionary. For example, if your context contained a dictionary `data`, the following would display the keys and values of the dictionary:

```
{% for key, value in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

The for loop sets a number of variables available within the loop:

Variable	Description
`forloop.counter`	The current iteration of the loop (1-indexed)
`forloop.counter0`	The current iteration of the loop (0-indexed)
`forloop.revcounter`	The number of iterations from the end of the loop (1-indexed)
`forloop.revcounter0`	The number of iterations from the end of the loop (0-indexed)
`forloop.first`	True if this is the first time through the loop
`forloop.last`	True if this is the last time through the loop
`forloop.parentloop`	For nested loops, this is the loop "above" the current one

for ... empty

The `for` tag can take an optional `{% empty %}` clause that will be displayed if the given array is empty or could not be found:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% empty %}
    <li>Sorry, no athlete in this list!</li>
{% endfor %}
<ul>
```

The above is equivalent to – but shorter, cleaner, and possibly faster than – the following:

```
<ul>
  {% if athlete_list %}
    {% for athlete in athlete_list %}
      <li>{{ athlete.name }}</li>
    {% endfor %}
```

```
  {% else %}
    <li>Sorry, no athletes in this list.</li>
  {% endif %}
</ul>
```

if

The `{% if %}` tag evaluates a variable, and if that variable is "true" (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% elif athlete_in_locker_room_list %}
    Athletes should be out of the locker room soon!
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable.

As you can see, the `if` tag may take one or several ‘‘ {% elif %}‘‘ clauses, as well as an `{% else %}` clause that will be displayed if all previous conditions fail. These clauses are optional. New in version 1.4: *Please see the release notes* The `if` tag now supports `{% elif %}` clauses.

Boolean operators

`if` tags may use `and`, `or` or `not` to test a number of variables or to negate a given variable:

```
{% if athlete_list and coach_list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete_list %}
    There are no athletes.
{% endif %}

{% if athlete_list or coach_list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete_list or coach_list %}
    There are no athletes or there are some coaches (OK, so
    writing English translations of boolean logic sounds
    stupid; it's not our fault).
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
{% endif %}
```

Use of both `and` and `or` clauses within the same tag is allowed, with `and` having higher precedence than `or` e.g.:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

will be interpreted like:

```
if (athlete_list and coach_list) or cheerleader_list
```

Use of actual parentheses in the `if` tag is invalid syntax. If you need them to indicate precedence, you should use nested `if` tags.

`if` tags may also use the operators ==, `!=`, <, >, <=, >= and `in` which work as follows:

== operator

Equality. Example:

```
{% if somevar == "x" %}
  This appears if variable somevar equals the string "x"
{% endif %}
```

!= operator

Inequality. Example:

```
{% if somevar != "x" %}
  This appears if variable somevar does not equal the string "x",
  or if somevar is not found in the context
{% endif %}
```

< operator

Less than. Example:

```
{% if somevar < 100 %}
  This appears if variable somevar is less than 100.
{% endif %}
```

> operator

Greater than. Example:

```
{% if somevar > 0 %}
  This appears if variable somevar is greater than 0.
{% endif %}
```

<= operator

Less than or equal to. Example:

```
{% if somevar <= 100 %}
  This appears if variable somevar is less than 100 or equal to 100.
{% endif %}
```

>= operator

Greater than or equal to. Example:

```
{% if somevar >= 1 %}
  This appears if variable somevar is greater than 1 or equal to 1.
{% endif %}
```

in operator

Contained within. This operator is supported by many Python containers to test whether the given value is in the container. The following are some examples of how `x in y` will be interpreted:

```
{% if "bc" in "abcdef" %}
  This appears since "bc" is a substring of "abcdef"
{% endif %}

{% if "hello" in greetings %}
  If greetings is a list or set, one element of which is the string
  "hello", this will appear.
{% endif %}

{% if user in users %}
  If users is a QuerySet, this will appear if user is an
  instance that belongs to the QuerySet.
{% endif %}
```

not in operator

Not contained within. This is the negation of the `in` operator.

The comparison operators cannot be 'chained' like in Python or in mathematical notation. For example, instead of using:

```
{% if a > b > c %}  (WRONG)
```

you should use:

```
{% if a > b and b > c %}
```

Filters

You can also use filters in the `if` expression. For example:

```
{% if messages|length >= 100 %}
   You have lots of messages today!
{% endif %}
```

Complex expressions

All of the above can be combined to form complex expressions. For such expressions, it can be important to know how the operators are grouped when the expression is evaluated - that is, the precedence rules. The precedence of the operators, from lowest to highest, is as follows:

- `or`
- `and`
- `not`
- `in`
- ==, !=, <, >, <=, >=

(This follows Python exactly). So, for example, the following complex `if` tag:

```
{% if a == b or c == d and e %}
```

...will be interpreted as:

```
(a == b) or ((c == d) and e)
```

If you need different precedence, you will need to use nested `if` tags. Sometimes that is better for clarity anyway, for the sake of those who do not know the precedence rules.

ifchanged

Check if a value has changed from the last iteration of a loop.

The `{% ifchanged %}` block tag is used within a loop. It has two possible uses.

1. Checks its own rendered contents against its previous state and only displays the content if it has changed. For example, this displays a list of days, only displaying the month if it changes:

```
<h1>Archive for {{ year }}</h1>

{% for date in days %}
    {% ifchanged %}<h3>{{ date|date:"F" }}</h3>{% endifchanged %}
    <a href="{{ date|date:"M/d"|lower }}/">{{ date|date:"j" }}</a>
{% endfor %}
```

2. If given one or more variables, check whether any variable has changed. For example, the following shows the date every time it changes, while showing the hour if either the hour or the date has changed:

```
{% for date in days %}
    {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
    {% ifchanged date.hour date.date %}
        {{ date.hour }}
    {% endifchanged %}
{% endfor %}
```

The `ifchanged` tag can also take an optional `{% else %}` clause that will be displayed if the value has not changed:

```
{% for match in matches %}
    <div style="background-color:
        {% ifchanged match.ballot_id %}
            {% cycle "red" "blue" %}
        {% else %}
            grey
        {% endifchanged %}
    ">{{ match }}</div>
{% endfor %}
```

ifequal

Output the contents of the block if the two arguments equal each other.

Example:

```
{% ifequal user.id comment.user_id %}
    ...
{% endifequal %}
```

As in the `if` tag, an `{% else %}` clause is optional.

The arguments can be hard-coded strings, so the following is valid:

```
{% ifequal user.username "adrian" %}
    ...
{% endifequal %}
```

It is only possible to compare an argument to template variables or strings. You cannot check for equality with Python objects such as `True` or `False`. If you need to test if something is true or false, use the `if` tag instead.

An alternative to the `ifequal` tag is to use the `if` tag and the `==` operator.

ifnotequal

Just like `ifequal`, except it tests that the two arguments are not equal.

An alternative to the `ifnotequal` tag is to use the `if` tag and the `!=` operator.

include

Loads a template and renders it with the current context. This is a way of "including" other templates within a template.

The template name can either be a variable or a hard-coded (quoted) string, in either single or double quotes.

This example includes the contents of the template `"foo/bar.html"`:

```
{% include "foo/bar.html" %}
```

This example includes the contents of the template whose name is contained in the variable `template_name`:

```
{% include template_name %}
```

An included template is rendered with the context of the template that's including it. This example produces the output `"Hello, John"`:

- Context: variable `person` is set to `"john"`.
- Template:

  ```
  {% include "name_snippet.html" %}
  ```

- The `name_snippet.html` template:

  ```
  {{ greeting }}, {{ person|default:"friend" }}!
  ```

Changed in version 1.3: Additional context and exclusive context. You can pass additional context to the template using keyword arguments:

```
{% include "name_snippet.html" with person="Jane" greeting="Hello" %}
```

If you want to only render the context with the variables provided (or even no variables at all), use the `only` option:

```
{% include "name_snippet.html" with greeting="Hi" only %}
```

Note: The `include` tag should be considered as an implementation of "render this subtemplate and include the HTML", not as "parse this subtemplate and include its contents as if it were part of the parent". This means that there is no shared state between included templates – each include is a completely independent rendering process.

See also: `{% ssi %}`.

load

Loads a custom template tag set.

For example, the following template would load all the tags and filters registered in `somelibrary` and `otherlibrary` located in package `package`:

```
{% load somelibrary package.otherlibrary %}
```

Changed in version 1.3: *Please see the release notes* You can also selectively load individual filters or tags from a library, using the `from` argument. In this example, the template tags/filters named `foo` and `bar` will be loaded from `somelibrary`:

```
{% load foo bar from somelibrary %}
```

See *Custom tag and filter libraries* for more information.

now

Displays the current date and/or time, using a format according to the given string. Such string can contain format specifiers characters as described in the `date` filter section.

Example:

```
It is {% now "jS F Y H:i" %}
```

Note that you can backslash-escape a format string if you want to use the "raw" value. In this example, "f" is backslash-escaped, because otherwise "f" is a format string that displays the time. The "o" doesn't need to be escaped, because it's not a format character:

```
It is the {% now "jS o\f F" %}
```

This would display as "It is the 4th of September". Changed in version 1.4: *Please see the release notes*

Note: The format passed can also be one of the predefined ones `DATE_FORMAT`, `DATETIME_FORMAT`, `SHORT_DATE_FORMAT` or `SHORT_DATETIME_FORMAT`. The predefined formats may vary depending on the current locale and if *Format localization* is enabled, e.g.:

```
It is {% now "SHORT_DATETIME_FORMAT" %}
```

---

regroup

Regroups a list of alike objects by a common attribute.

This complex tag is best illustrated by way of an example: say that "places" is a list of cities represented by dictionaries containing `"name"`, `"population"`, and `"country"` keys:

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'India'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country': 'India'},
    {'name': 'New York', 'population': '20,000,000', 'country': 'USA'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'USA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Japan'},
]
```

...and you'd like to display a hierarchical list that is ordered by country, like this:

- **India**
  - Mumbai: 19,000,000
  - Calcutta: 15,000,000
- **USA**
  - New York: 20,000,000
  - Chicago: 7,000,000
- **Japan**
  - Tokyo: 33,000,000

You can use the `{% regroup %}` tag to group the list of cities by country. The following snippet of template code would accomplish this:

```
{% regroup cities by country as country_list %}

<ul>
{% for country in country_list %}
    <li>{{ country.grouper }}
    <ul>
        {% for item in country.list %}
          <li>{{ item.name }}: {{ item.population }}</li>
        {% endfor %}
    </ul>
    </li>
{% endfor %}
</ul>
```

Let's walk through this example. `{% regroup %}` takes three arguments: the list you want to regroup, the attribute to group by, and the name of the resulting list. Here, we're regrouping the `cities` list by the `country` attribute and calling the result `country_list`.

`{% regroup %}` produces a list (in this case, `country_list`) of **group objects**. Each group object has two attributes:

- `grouper` – the item that was grouped by (e.g., the string "India" or "Japan").
- `list` – a list of all items in this group (e.g., a list of all cities with country='India').

Note that `{% regroup %}` does not order its input! Our example relies on the fact that the `cities` list was ordered by `country` in the first place. If the `cities` list did *not* order its members by `country`, the regrouping would

naively display more than one group for a single country. For example, say the `cities` list was set to this (note that the countries are not grouped together):

```
cities = [
    {'name': 'Mumbai', 'population': '19,000,000', 'country': 'India'},
    {'name': 'New York', 'population': '20,000,000', 'country': 'USA'},
    {'name': 'Calcutta', 'population': '15,000,000', 'country': 'India'},
    {'name': 'Chicago', 'population': '7,000,000', 'country': 'USA'},
    {'name': 'Tokyo', 'population': '33,000,000', 'country': 'Japan'},
]
```

With this input for `cities`, the example `{% regroup %}` template code above would result in the following output:

- **India**
  - Mumbai: 19,000,000
- **USA**
  - New York: 20,000,000
- **India**
  - Calcutta: 15,000,000
- **Japan**
  - Tokyo: 33,000,000

The easiest solution to this gotcha is to make sure in your view code that the data is ordered according to how you want to display it.

Another solution is to sort the data in the template using the `dictsort` filter, if your data is in a list of dictionaries:

```
{% regroup cities|dictsort:"country" by country as country_list %}
```

Grouping on other properties

Any valid template lookup is a legal grouping attribute for the regroup tag, including methods, attributes, dictionary keys and list items. For example, if the "country" field is a foreign key to a class with an attribute "description," you could use:

```
{% regroup cities by country.description as country_list %}
```

Or, if `country` is a field with `choices`, it will have a `^django.db.models.Model.get_FOO_display()` method available as an attribute, allowing you to group on the display string rather than the `choices` key:

```
{% regroup cities by get_country_display as country_list %}
```

`{{ country.grouper }}` will now display the value fields from the `choices` set rather than the keys.

spaceless

Removes whitespace between HTML tags. This includes tab characters and newlines.

Example usage:

```
{% spaceless %}
    <p>
        <a href="foo/">Foo</a>
    </p>
{% endspaceless %}
```

This example would return this HTML:

```
<p><a href="foo/">Foo</a></p>
```

Only space between *tags* is removed – not space between tags and text. In this example, the space around `Hello` won't be stripped:

```
{% spaceless %}
    <strong>
        Hello
    </strong>
{% endspaceless %}
```

ssi

Outputs the contents of a given file into the page.

Like a simple `include` tag, `{% ssi %}` includes the contents of another file – which must be specified using an absolute path – in the current page:

```
{% ssi '/home/html/ljworld.com/includes/right_generic.html' %}
```

The first parameter of `ssi` can be a quoted literal or any other context variable.

If the optional "parsed" parameter is given, the contents of the included file are evaluated as template code, within the current context:

```
{% ssi '/home/html/ljworld.com/includes/right_generic.html' parsed %}
```

Note that if you use `{% ssi %}`, you'll need to define `ALLOWED_INCLUDE_ROOTS` in your Django settings, as a security measure.

See also: `{% include %}`.

templatetag

Outputs one of the syntax characters used to compose template tags.

Since the template system has no concept of "escaping", to display one of the bits used in template tags, you must use the `{% templatetag %}` tag.

The argument tells which template bit to output:

Argument	Outputs
`openblock`	`{%`
`closeblock`	`%}`
`openvariable`	`{{`
`closevariable`	`}}`
`openbrace`	`{`
`closebrace`	`}`
`opencomment`	`{#`
`closecomment`	`#}`

url

Returns an absolute path reference (a URL without the domain name) matching a given view function and optional parameters. This is a way to output links without violating the DRY principle by having to hard-code URLs in your templates:

```
{% url 'path.to.some_view' v1 v2 %}
```

The first argument is a path to a view function in the format `package.package.module.function`. It can be a quoted literal or any other context variable. Additional arguments are optional and should be space-separated values that will be used as arguments in the URL. The example above shows passing positional arguments. Alternatively you may use keyword syntax:

```
{% url 'path.to.some_view' arg1=v1 arg2=v2 %}
```

Do not mix both positional and keyword syntax in a single call. All arguments required by the URLconf should be present.

For example, suppose you have a view, `app_views.client`, whose URLconf takes a client ID (here, `client()` is a method inside the views file `app_views.py`). The URLconf line might look like this:

```
('^client/(\d+)/$', 'app_views.client')
```

If this app's URLconf is included into the project's URLconf under a path such as this:

```
('^clients/', include('project_name.app_name.urls'))
```

...then, in a template, you can create a link to this view like this:

```
{% url 'app_views.client' client.id %}
```

The template tag will output the string `/clients/client/123/`.

If you're using *named URL patterns*, you can refer to the name of the pattern in the `url` tag instead of using the path to the view.

Note that if the URL you're reversing doesn't exist, you'll get an `^django.core.urlresolvers.NoReverseMatch` exception raised, which will cause your site to display an error page.

If you'd like to retrieve a URL without displaying it, you can use a slightly different call:

```
{% url 'path.to.view' arg arg2 as the_url %}

<a href="{{ the_url }}">I'm linking to {{ the_url }}</a>
```

This `{% url ... as var %}` syntax will *not* cause an error if the view is missing. In practice you'll use this to link to views that are optional:

```
{% url 'path.to.view' as the_url %}
{% if the_url %}
  <a href="{{ the_url }}">Link to optional stuff</a>
{% endif %}
```

If you'd like to retrieve a namespaced URL, specify the fully qualified name:

```
{% url 'myapp:view-name' %}
```

This will follow the normal *namespaced URL resolution strategy*, including using any hints provided by the context as to the current application.

verbatim

New in version 1.5: *Please see the release notes* Stops the template engine from rendering the contents of this block tag.

A common use is to allow a Javascript template layer that collides with Django's syntax. For example:

```
{% verbatim %}
    {{if dying}}Still alive.{{/if}}
{% endverbatim %}
```

You can also designate a specific closing tag, allowing the use of `{% endverbatim %}` as part of the unrendered contents:

```
{% verbatim myblock %}
    Avoid template rendering via the {% verbatim %}{% endverbatim %} block.
{% endverbatim myblock %}
```

widthratio

For creating bar charts and such, this tag calculates the ratio of a given value to a maximum value, and then applies that ratio to a constant.

For example:

```
<img src="bar.png" alt="Bar"
     height="10" width="{% widthratio this_value max_value 100 %}" />
```

Above, if `this_value` is 175 and `max_value` is 200, the image in the above example will be 88 pixels wide (because 175/200 = .875; .875 * 100 = 87.5 which is rounded up to 88).

with

Changed in version 1.3: New keyword argument format and multiple variable assignments. Caches a complex variable under a simpler name. This is useful when accessing an "expensive" method (e.g., one that hits the database) multiple times.

For example:

```
{% with total=business.employees.count %}
    {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

The populated variable (in the example above, `total`) is only available between the `{% with %}` and `{% endwith %}` tags.

You can assign more than one context variable:

```
{% with alpha=1 beta=2 %}
    ...
{% endwith %}
```

Note: The previous more verbose format is still supported: `{% with business.employees.count as total %}`

Built-in filter reference

add

Adds the argument to the value.

For example:

```
{{ value|add:"2" }}
```

If `value` is `4`, then the output will be `6`.

This filter will first try to coerce both values to integers. If this fails, it'll attempt to add the values together anyway. This will work on some data types (strings, list, etc.) and fail on others. If it fails, the result will be an empty string.

For example, if we have:

```
{{ first|add:second }}
```

and `first` is `[1, 2, 3]` and `second` is `[4, 5, 6]`, then the output will be `[1, 2, 3, 4, 5, 6]`.

> **Warning:** Strings that can be coerced to integers will be **summed**, not concatenated, as in the first example above.

addslashes

Adds slashes before quotes. Useful for escaping strings in CSV, for example.

For example:

```
{{ value|addslashes }}
```

If `value` is `"I'm using Django"`, the output will be `"I\'m using Django"`.

capfirst

Capitalizes the first character of the value.

For example:

```
{{ value|capfirst }}
```

If `value` is `"django"`, the output will be `"Django"`.

center

Centers the value in a field of a given width.

For example:

```
"{{ value|center:"15" }}"
```

If `value` is `"Django"`, the output will be `" Django "`.

cut

Removes all values of arg from the given string.

For example:

```
{{ value|cut:" "}}
```

If `value` is `"String with spaces"`, the output will be `"Stringwithspaces"`.

date

Formats a date according to the given format.

Uses a similar format as PHP's `date()` function (http://php.net/date) with some differences.

Available format strings:

Format character	Description	Example output
a	`'a.m.'` or `'p.m.'` (Note that this is slightly different than PHP's output, because this includes periods to m
A	`'AM'` or `'PM'`.
b	Month, textual, 3 letters, lowercase.
B	Not implemented.
c	ISO 8601 format. (Note: unlike others formatters, such as "Z", "O" or "r", the "c" formatter will not add timez
d	Day of the month, 2 digits with leading zeros.
D	Day of the week, textual, 3 letters.
e	Timezone name. Could be in any format, or might return an empty string, depending on the datetime.
E	Month, locale specific alternative representation usually used for long date representation.
f	Time, in 12-hour hours and minutes, with minutes left off if they're zero. Proprietary extension.
F	Month, textual, long.
g	Hour, 12-hour format without leading zeros.
G	Hour, 24-hour format without leading zeros.
h	Hour, 12-hour format.
H	Hour, 24-hour format.
i	Minutes.
I	Not implemented.
j	Day of the month without leading zeros.
l	Day of the week, textual, long.
L	Boolean for whether it's a leap year.
m	Month, 2 digits with leading zeros.
M	Month, textual, 3 letters.
n	Month without leading zeros.
N	Month abbreviation in Associated Press style. Proprietary extension.
o	ISO-8601 week-numbering year, corresponding to the ISO-8601 week number (W)
O	Difference to Greenwich time in hours.
P	Time, in 12-hour hours, minutes and 'a.m.'/'p.m.', with minutes left off if they're zero and the special-case stri
r	**RFC 2822** formatted date.
s	Seconds, 2 digits with leading zeros.
S	English ordinal suffix for day of the month, 2 characters.
t	Number of days in the given month.
T	Time zone of this machine.
u	Microseconds.

Continued on next page

Table 6.4 – continued from previous page

Format character	Description	Example output
U	Seconds since the Unix Epoch (January 1 1970 00:00:00 UTC).	
w	Day of the week, digits without leading zeros.	
W	ISO-8601 week number of year, with weeks starting on Monday.	
y	Year, 2 digits.	
Y	Year, 4 digits.	
z	Day of the year.	
Z	Time zone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UT	

New in version 1.4: *Please see the release notes* The `e` and `o` format specification characters were added in Django 1.4.

For example:

```
{{ value|date:"D d M Y" }}
```

If `value` is a `datetime` object (e.g., the result of `datetime.datetime.now()`), the output will be the string `'Wed 09 Jan 2008'`.

The format passed can be one of the predefined ones `DATE_FORMAT`, `DATETIME_FORMAT`, `SHORT_DATE_FORMAT` or `SHORT_DATETIME_FORMAT`, or a custom format that uses the format specifiers shown in the table above. Note that predefined formats may vary depending on the current locale.

Assuming that `USE_L10N` is `True` and `LANGUAGE_CODE` is, for example, `"es"`, then for:

```
{{ value|date:"SHORT_DATE_FORMAT" }}
```

the output would be the string `"09/01/2008"` (the `"SHORT_DATE_FORMAT"` format specifier for the `es` locale as shipped with Django is `"d/m/Y"`).

When used without a format string:

```
{{ value|date }}
```

...the formatting string defined in the `DATE_FORMAT` setting will be used, without applying any localization.

default

If value evaluates to `False`, uses the given default. Otherwise, uses the value.

For example:

```
{{ value|default:"nothing" }}
```

If `value` is `""` (the empty string), the output will be `nothing`.

default_if_none

If (and only if) value is `None`, uses the given default. Otherwise, uses the value.

Note that if an empty string is given, the default value will *not* be used. Use the `default` filter if you want to fallback for empty strings.

For example:

```
{{ value|default_if_none:"nothing" }}
```

If `value` is `None`, the output will be the string `"nothing"`.

dictsort

Takes a list of dictionaries and returns that list sorted by the key given in the argument.

For example:

```
{{ value|dictsort:"name" }}
```

If `value` is:

```
[
    {'name': 'zed', 'age': 19},
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
]
```

then the output would be:

```
[
    {'name': 'amy', 'age': 22},
    {'name': 'joe', 'age': 31},
    {'name': 'zed', 'age': 19},
]
```

dictsortreversed

Takes a list of dictionaries and returns that list sorted in reverse order by the key given in the argument. This works exactly the same as the above filter, but the returned value will be in reverse order.

divisibleby

Returns `True` if the value is divisible by the argument.

For example:

```
{{ value|divisibleby:"3" }}
```

If `value` is `21`, the output would be `True`.

escape

Escapes a string's HTML. Specifically, it makes these replacements:

- < is converted to `<`
- > is converted to `>`
- ' (single quote) is converted to `&#39;`
- " (double quote) is converted to `&quot;`
- & is converted to `&`

The escaping is only applied when the string is output, so it does not matter where in a chained sequence of filters you put `escape`: it will always be applied as though it were the last filter. If you want escaping to be applied immediately, use the `force_escape` filter.

Applying `escape` to a variable that would normally have auto-escaping applied to the result will only result in one round of escaping being done. So it is safe to use this function even in auto-escaping environments. If you want multiple escaping passes to be applied, use the `force_escape` filter.

escapejs

Escapes characters for use in JavaScript strings. This does *not* make the string safe for use in HTML, but does protect you from syntax errors when using templates to generate JavaScript/JSON.

For example:

```
{{ value|escapejs }}
```

If `value` is `"testing\r\njavascript \'string" <b>escaping</b>"`, the output will be `"testing\\u000D\\u000Ajavascript \\u0027string\\u0022 \\u003Cb\\u003Eescaping\\u003C/b\\u003E"`.

filesizeformat

Formats the value like a 'human-readable' file size (i.e. `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, etc).

For example:

```
{{ value|filesizeformat }}
```

If `value` is 123456789, the output would be `117.7 MB`.

first

Returns the first item in a list.

For example:

```
{{ value|first }}
```

If `value` is the list `['a', 'b', 'c']`, the output will be `'a'`.

fix_ampersands

Note: This is rarely useful as ampersands are automatically escaped. See `escape` for more information.

Replaces ampersands with `&` entities.

For example:

```
{{ value|fix_ampersands }}
```

If `value` is `Tom & Jerry`, the output will be `Tom &amp; Jerry`.

However, ampersands used in named entities and numeric character references will not be replaced. For example, if `value` is `Caf&eacute;`, the output will *not* be `Caf&eacute;` but remain `Caf&eacute;`. This means that in some edge cases, such as acronyms followed by semicolons, this filter will not replace ampersands that need replacing. For example, if `value` is `Contact the R&D;`, the output will remain unchanged because `&D;` resembles a named entity.

floatformat

When used without an argument, rounds a floating-point number to one decimal place – but only if there's a decimal part to be displayed. For example:

value	Template	Output
34.23234	{{ value|floatformat }}	34.2
34.00000	{{ value|floatformat }}	34
34.26000	{{ value|floatformat }}	34.3

If used with a numeric integer argument, `floatformat` rounds a number to that many decimal places. For example:

value	Template	Output
34.23234	{{ value|floatformat:3 }}	34.232
34.00000	{{ value|floatformat:3 }}	34.000
34.26000	{{ value|floatformat:3 }}	34.260

If the argument passed to `floatformat` is negative, it will round a number to that many decimal places – but only if there's a decimal part to be displayed. For example:

value	Template	Output
34.23234	{{ value|floatformat:"-3" }}	34.232
34.00000	{{ value|floatformat:"-3" }}	34
34.26000	{{ value|floatformat:"-3" }}	34.260

Using `floatformat` with no argument is equivalent to using `floatformat` with an argument of `-1`.

force_escape

Applies HTML escaping to a string (see the `escape` filter for details). This filter is applied *immediately* and returns a new, escaped string. This is useful in the rare cases where you need multiple escaping or want to apply other filters to the escaped results. Normally, you want to use the `escape` filter.

get_digit

Given a whole number, returns the requested digit, where 1 is the right-most digit, 2 is the second-right-most digit, etc. Returns the original value for invalid input (if input or argument is not an integer, or if argument is less than 1). Otherwise, output is always an integer.

For example:

```
{{ value|get_digit:"2" }}
```

If `value` is `123456789`, the output will be `8`.

iriencode

Converts an IRI (Internationalized Resource Identifier) to a string that is suitable for including in a URL. This is necessary if you're trying to use strings containing non-ASCII characters in a URL.

It's safe to use this filter on a string that has already gone through the `urlencode` filter.

For example:

```
{{ value|iriencode }}
```

If `value` is `"?test=1&me=2"`, the output will be `"?test=1&me=2"`.

join

Joins a list with a string, like Python's `str.join(list)`

For example:

```
{{ value|join:" // " }}
```

If `value` is the list `['a', 'b', 'c']`, the output will be the string `"a // b // c"`.

last

Returns the last item in a list.

For example:

```
{{ value|last }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output will be the string `"d"`.

length

Returns the length of the value. This works for both strings and lists.

For example:

```
{{ value|length }}
```

If `value` is `['a', 'b', 'c', 'd']`, the output will be `4`.

length_is

Returns `True` if the value's length is the argument, or `False` otherwise.

For example:

```
{{ value|length_is:"4" }}
```

If `value` is `['a', 'b', 'c', 'd']`, the output will be `True`.

linebreaks

Replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break (`<br />`) and a new line followed by a blank line becomes a paragraph break (`</p>`).

For example:

```
{{ value|linebreaks }}
```

If `value` is `Joel\nis a slug`, the output will be `<p>Joel<br />is a slug</p>`.

linebreaksbr

Converts all newlines in a piece of plain text to HTML line breaks (`<br />`).

For example:

```
{{ value|linebreaksbr }}
```

If `value` is `Joel\nis a slug`, the output will be `Joel<br />is a slug`.

linenumbers

Displays text with line numbers.

For example:

```
{{ value|linenumbers }}
```

If `value` is:

```
one
two
three
```

the output will be:

```
1. one
2. two
3. three
```

ljust

Left-aligns the value in a field of a given width.

Argument: field size

For example:

```
"{{ value|ljust:"10" }}"
```

If `value` is `Django`, the output will be `"Django "`.

lower

Converts a string into all lowercase.

For example:

```
{{ value|lower }}
```

If `value` is `Still MAD At Yoko`, the output will be `still mad at yoko`.

make_list

Returns the value turned into a list. For a string, it's a list of characters. For an integer, the argument is cast into an unicode string before creating a list.

For example:

```
{{ value|make_list }}
```

If `value` is the string `"Joel"`, the output would be the list `[u'J', u'o', u'e', u'l']`. If `value` is `123`, the output will be the list `[u'1', u'2', u'3']`.

phone2numeric

Converts a phone number (possibly containing letters) to its numerical equivalent.

The input doesn't have to be a valid phone number. This will happily convert any string.

For example:

```
{{ value|phone2numeric }}
```

If `value` is `800-COLLECT`, the output will be `800-2655328`.

pluralize

Returns a plural suffix if the value is not 1. By default, this suffix is `'s'`.

Example:

```
You have {{ num_messages }} message{{ num_messages|pluralize }}.
```

If `num_messages` is `1`, the output will be `You have 1 message.` If `num_messages` is `2` the output will be `You have 2 messages.`

For words that require a suffix other than `'s'`, you can provide an alternate suffix as a parameter to the filter.

Example:

```
You have {{ num_walruses }} walrus{{ num_walruses|pluralize:"es" }}.
```

For words that don't pluralize by simple suffix, you can specify both a singular and plural suffix, separated by a comma.

Example:

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

Note: Use `blocktrans` to pluralize translated strings.

---

pprint

A wrapper around `pprint.pprint()` – for debugging, really.

random

Returns a random item from the given list.

For example:

```
{{ value|random }}
```

If `value` is the list `['a', 'b', 'c', 'd']`, the output could be `"b"`.

removetags

Removes a space-separated list of [X]HTML tags from the output.

For example:

```
{{ value|removetags:"b span"|safe }}
```

If `value` is `"<b>Joel</b> <button>is</button> a <span>slug</span>"` the output will be `"Joel <button>is</button> a slug"`.

Note that this filter is case-sensitive.

If `value` is `"<B>Joel</B> <button>is</button> a <span>slug</span>"` the output will be `"<B>Joel</B> <button>is</button> a slug"`.

rjust

Right-aligns the value in a field of a given width.

Argument: field size

For example:

```
"{{ value|rjust:"10" }}"
```

If `value` is `Django`, the output will be `" Django"`.

safe

Marks a string as not requiring further HTML escaping prior to output. When autoescaping is off, this filter has no effect.

Note: If you are chaining filters, a filter applied after `safe` can make the contents unsafe again. For example, the following code prints the variable as is, unescaped:

```
{{ var|safe|escape }}
```

---

safeseq

Applies the `safe` filter to each element of a sequence. Useful in conjunction with other filters that operate on sequences, such as `join`. For example:

```
{{ some_list|safeseq|join:", " }}
```

You couldn't use the `safe` filter directly in this case, as it would first convert the variable into a string, rather than working with the individual elements of the sequence.

slice

Returns a slice of the list.

Uses the same syntax as Python's list slicing. See http://diveintopython.net/native_data_types/lists.html#odbchelper.list.slice for an introduction.

Example:

```
{{ some_list|slice:":2" }}
```

If `some_list` is `['a', 'b', 'c']`, the output will be `['a', 'b']`.

slugify

Converts to lowercase, removes non-word characters (alphanumerics and underscores) and converts spaces to hyphens. Also strips leading and trailing whitespace.

For example:

```
{{ value|slugify }}
```

If `value` is `"Joel is a slug"`, the output will be `"joel-is-a-slug"`.

stringformat

Formats the variable according to the argument, a string formatting specifier. This specifier uses Python string formatting syntax, with the exception that the leading "%" is dropped.

See http://docs.python.org/library/stdtypes.html#string-formatting-operations for documentation of Python string formatting

For example:

```
{{ value|stringformat:"s" }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel is a slug"`.

striptags

Strips all [X]HTML tags.

For example:

```
{{ value|striptags }}
```

If `value` is `"<b>Joel</b> <button>is</button> a <span>slug</span>"`, the output will be `"Joel is a slug"`.

time

Formats a time according to the given format.

Given format can be the predefined one `TIME_FORMAT`, or a custom format, same as the `date` filter. Note that the predefined format is locale-dependant.

The time filter will only accept parameters in the format string that relate to the time of day, not the date (for obvious reasons). If you need to format a date, use the `date` filter.

For example:

```
{{ value|time:"H:i" }}
```

If `value` is equivalent to `datetime.datetime.now()`, the output will be the string `"01:23"`.

Another example:

Assuming that `USE_L10N` is `True` and `LANGUAGE_CODE` is, for example, `"de"`, then for:

```
{{ value|time:"TIME_FORMAT" }}
```

the output will be the string `"01:23:00"` (The `"TIME_FORMAT"` format specifier for the `de` locale as shipped with Django is `"H:i:s"`).

When used without a format string:

```
{{ value|time }}
```

...the formatting string defined in the `TIME_FORMAT` setting will be used, without applying any localization.

timesince

Formats a date as the time since that date (e.g., “4 days, 6 hours”).

Takes an optional argument that is a variable containing the date to use as the comparison point (without the argument, the comparison point is *now*). For example, if `blog_date` is a date instance representing midnight on 1 June 2006, and `comment_date` is a date instance for 08:00 on 1 June 2006, then `{{ blog_date|timesince:comment_date }}` would return “8 hours”.

Comparing offset-naive and offset-aware datetimes will return an empty string.

Minutes is the smallest unit used, and “0 minutes” will be returned for any date that is in the future relative to the comparison point.

timeuntil

Similar to `timesince`, except that it measures the time from now until the given date or datetime. For example, if today is 1 June 2006 and `conference_date` is a date instance holding 29 June 2006, then `{{ conference_date|timeuntil }}` will return "4 weeks".

Takes an optional argument that is a variable containing the date to use as the comparison point (instead of *now*). If `from_date` contains 22 June 2006, then `{{ conference_date|timeuntil:from_date }}` will return "1 week".

Comparing offset-naive and offset-aware datetimes will return an empty string.

Minutes is the smallest unit used, and "0 minutes" will be returned for any date that is in the past relative to the comparison point.

title

Converts a string into titlecase.

For example:

```
{{ value|title }}
```

If `value` is `"my first post"`, the output will be `"My First Post"`.

truncatechars

New in version 1.4: *Please see the release notes* Truncates a string if it is longer than the specified number of characters. Truncated strings will end with a translatable ellipsis sequence ("...").

Argument: Number of characters to truncate to

For example:

```
{{ value|truncatechars:9 }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel i..."`.

truncatewords

Truncates a string after a certain number of words.

Argument: Number of words to truncate after

For example:

```
{{ value|truncatewords:2 }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel is ..."`.

Newlines within the string will be removed.

truncatewords_html

Similar to `truncatewords`, except that it is aware of HTML tags. Any tags that are opened in the string and not closed before the truncation point, are closed immediately after the truncation.

This is less efficient than `truncatewords`, so should only be used when it is being passed HTML text.

For example:

```
{{ value|truncatewords_html:2 }}
```

If `value` is `"<p>Joel is a slug</p>"`, the output will be `"<p>Joel is ...</p>"`.

Newlines in the HTML content will be preserved.

unordered_list

Recursively takes a self-nested list and returns an HTML unordered list – WITHOUT opening and closing <ul> tags.

The list is assumed to be in the proper format. For example, if `var` contains `['States', ['Kansas', ['Lawrence', 'Topeka'], 'Illinois']]`, then `{{ var|unordered_list }}` would return:

```
<li>States
<ul>
        <li>Kansas
        <ul>
                <li>Lawrence</li>
                <li>Topeka</li>
        </ul>
        </li>
        <li>Illinois</li>
</ul>
</li>
```

Note: An older, more restrictive and verbose input format is also supported: `['States', [['Kansas', [['Lawrence', []], ['Topeka', []]]], ['Illinois', []]]]`.

upper

Converts a string into all uppercase.

For example:

```
{{ value|upper }}
```

If `value` is `"Joel is a slug"`, the output will be `"JOEL IS A SLUG"`.

urlencode

Escapes a value for use in a URL.

For example:

```
{{ value|urlencode }}
```

If `value` is `"http://www.example.org/foo?a=b&c=d"`, the output will be `"http%3A//www.example.org/foo%3Fa%3Db%26c%3Dd"`. New in version 1.3: *Please see the release notes* An optional argument containing the characters which should not be escaped can be provided.

If not provided, the '/' character is assumed safe. An empty string can be provided when *all* characters should be escaped. For example:

```
{{ value|urlencode:"" }}
```

If `value` is `"http://www.example.org/"`, the output will be `"http%3A%2F%2Fwww.example.org%2F"`.

urlize

Converts URLs in text into clickable links.

This template tag works on links prefixed with `http://`, `https://`, or `www.`. For example, `http://goo.gl/aia1t` will get converted but `goo.gl/aia1t` won't.

It also supports domain-only links ending in one of the original top level domains (`.com`, `.edu`, `.gov`, `.int`, `.mil`, `.net`, and `.org`). For example, `djangoproject.com` gets converted. Changed in version 1.4: *Please see the release notes* Until Django 1.4, only the `.com`, `.net` and `.org` suffixes were supported for domain-only links.

Links can have trailing punctuation (periods, commas, close-parens) and leading punctuation (opening parens), and `urlize` will still do the right thing.

Links generated by `urlize` have a `rel="nofollow"` attribute added to them.

For example:

```
{{ value|urlize }}
```

If `value` is `"Check out www.djangoproject.com"`, the output will be `"Check out <a href="http://www.djangoproject.com" rel="nofollow">www.djangoproject.com</a>"`.

The `urlize` filter also takes an optional parameter `autoescape`. If `autoescape` is `True`, the link text and URLs will be escaped using Django's built-in `escape` filter. The default value for `autoescape` is `True`.

Note: If `urlize` is applied to text that already contains HTML markup, things won't work as expected. Apply this filter only to plain text.

---

urlizetrunc

Converts URLs into clickable links just like urlize, but truncates URLs longer than the given character limit.

Argument: Number of characters that link text should be truncated to, including the ellipsis that's added if truncation is necessary.

For example:

```
{{ value|urlizetrunc:15 }}
```

If `value` is `"Check out www.djangoproject.com"`, the output would be `'Check out <a href="http://www.djangoproject.com" rel="nofollow">www.djangopr...</a>'`.

As with urlize, this filter should only be applied to plain text.

wordcount

Returns the number of words.

For example:

```
{{ value|wordcount }}
```

If `value` is `"Joel is a slug"`, the output will be `4`.

wordwrap

Wraps words at specified line length.

Argument: number of characters at which to wrap the text

For example:

```
{{ value|wordwrap:5 }}
```

If `value` is `Joel is a slug`, the output would be:

```
Joel
is a
slug
```

yesno

Maps values for true, false and (optionally) None, to the strings “yes”, “no”, “maybe”, or a custom mapping passed as a comma-separated list, and returns one of those strings according to the value:

For example:

```
{{ value|yesno:"yeah,no,maybe" }}
```

Value	Argument	Outputs
`True`		`yes`
`True`	`"yeah,no,maybe"`	`yeah`
`False`	`"yeah,no,maybe"`	`no`
`None`	`"yeah,no,maybe"`	`maybe`
`None`	`"yeah,no"`	`"no"` (converts None to False if no mapping for None is given)

Internationalization tags and filters

Django provides template tags and filters to control each aspect of *internationalization* in templates. They allow for granular control of translations, formatting, and time zone conversions.

i18n

This library allows specifying translatable text in templates. To enable it, set `USE_I18N` to `True`, then load it with `{% load i18n %}`.

See *Internationalization: in template code*.

l10n

This library provides control over the localization of values in templates. You only need to load the library using `{% load l10n %}`, but you'll often set `USE_L10N` to `True` so that localization is active by default.

See *Controlling localization in templates*.

tz

New in version 1.4: *Please see the release notes* This library provides control over time zone conversions in templates. Like `l10n`, you only need to load the library using `{% load tz %}`, but you'll usually also set `USE_TZ` to `True` so that conversion to local time happens by default.

See *Time zone aware output in templates*.

Other tags and filters libraries

Django comes with a couple of other template-tag libraries that you have to enable explicitly in your `INSTALLED_APPS` setting and enable in your template with the `{% load %}` tag.

django.contrib.humanize

A set of Django template filters useful for adding a "human touch" to data. See *django.contrib.humanize*.

django.contrib.markup

A collection of template filters that implement these common markup languages:

- Textile
- Markdown
- reST (reStructuredText)

See the *markup documentation*.

django.contrib.webdesign

A collection of template tags that can be useful while designing a Web site, such as a generator of Lorem Ipsum text. See *django.contrib.webdesign*.

static

static To link to static files that are saved in `STATIC_ROOT` Django ships with a `static` template tag. You can use this regardless if you're using `RequestContext` or not.

```
{% load static %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

It is also able to consume standard context variables, e.g. assuming a `user_stylesheet` variable is passed to the template:

```
{% load static %}
<link rel="stylesheet" href="{% static user_stylesheet %}" type="text/css" media="screen" />
```

If you'd like to retrieve a static URL without displaying it, you can use a slightly different call:

```
.. versionadded:: 1.5
```

```
{% load static %}
{% static "images/hi.jpg" as myphoto %}
<img src="{{ myphoto }}"></img>
```

Note: The `staticfiles` contrib app also ships with a `static template tag` which uses `staticfiles'` `STATICFILES_STORAGE` to build the URL of the given path. Use that instead if you have an advanced use case such as *using a cloud service to serve static files*:

```
{% load static from staticfiles %}
<img src="{% static "images/hi.jpg" %}" alt="Hi!" />
```

---

get_static_prefix If you're not using `RequestContext`, or if you need more control over exactly where and how `STATIC_URL` is injected into the template, you can use the `get_static_prefix` template tag instead:

```
{% load static %}
<img src="{% get_static_prefix %}images/hi.jpg" alt="Hi!" />
```

There's also a second form you can use to avoid extra processing if you need the value multiple times:

```
{% load static %}
{% get_static_prefix as STATIC_PREFIX %}

<img src="{{ STATIC_PREFIX }}images/hi.jpg" alt="Hi!" />
<img src="{{ STATIC_PREFIX }}images/hi2.jpg" alt="Hello!" />
```

get_media_prefix Similar to the `get_static_prefix`, `get_media_prefix` populates a template variable with the media prefix `MEDIA_URL`, e.g.:

```
<script type="text/javascript" charset="utf-8">
var media_path = '{% get_media_prefix %}';
</script>
```

6.17.2 The Django template language: For Python programmers

This document explains the Django template system from a technical perspective – how it works and how to extend it. If you're just looking for reference on the language syntax, see *The Django template language*.

If you're looking to use the Django template system as part of another application – i.e., without the rest of the framework – make sure to read the configuration section later in this document.

Basics

A **template** is a text document, or a normal Python string, that is marked-up using the Django template language. A template can contain **block tags** or **variables**.

A **block tag** is a symbol within a template that does something.

This definition is deliberately vague. For example, a block tag can output content, serve as a control structure (an "if" statement or "for" loop), grab content from a database or enable access to other template tags.

Block tags are surrounded by `"{%"` and `"%}"`.

Example template with block tags:

```
{% if is_logged_in %}Thanks for logging in!{% else %}Please log in.{% endif %}
```

A **variable** is a symbol within a template that outputs a value.

Variable tags are surrounded by `"{{"` and `"}}"`.

Example template with variables:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

A **context** is a "variable name" -> "variable value" mapping that is passed to a template.

A template **renders** a context by replacing the variable "holes" with values from the context and executing all block tags.

Using the template system

class `django.template.`**`Template`**

Using the template system in Python is a two-step process:

- First, you compile the raw template code into a `Template` object.
- Then, you call the `render()` method of the `Template` object with a given context.

Compiling a string

The easiest way to create a `Template` object is by instantiating it directly. The class lives at `django.template.Template`. The constructor takes one argument – the raw template code:

```
>>> from django.template import Template
>>> t = Template("My name is {{ my_name }}.")
>>> print(t)
<django.template.Template instance>
```

Behind the scenes

The system only parses your raw template code once – when you create the `Template` object. From then on, it's stored internally as a "node" structure for performance.

Even the parsing itself is quite fast. Most of the parsing happens via a single call to a single, short, regular expression.

Rendering a context

`render`(*context*)

Once you have a compiled `Template` object, you can render a context – or multiple contexts – with it. The `Context` class lives at `django.template.Context`, and the constructor takes two (optional) arguments:

- A dictionary mapping variable names to variable values.

- The name of the current application. This application name is used to help *resolve namespaced URLs*. If you're not using namespaced URLs, you can ignore this argument.

Call the `Template` object's `render()` method with the context to "fill" the template:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ my_name }}.")

>>> c = Context({"my_name": "Adrian"})
>>> t.render(c)
"My name is Adrian."

>>> c = Context({"my_name": "Dolores"})
>>> t.render(c)
"My name is Dolores."
```

Variables and lookups Variable names must consist of any letter (A-Z), any digit (0-9), an underscore (but they must not start with an underscore) or a dot.

Dots have a special meaning in template rendering. A dot in a variable name signifies a **lookup**. Specifically, when the template system encounters a dot in a variable name, it tries the following lookups, in this order:

- Dictionary lookup. Example: `foo["bar"]`
- Attribute lookup. Example: `foo.bar`
- List-index lookup. Example: `foo[bar]`

The template system uses the first lookup type that works. It's short-circuit logic. Here are a few examples:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ person.first_name }}.")
>>> d = {"person": {"first_name": "Joe", "last_name": "Johnson"}}
>>> t.render(Context(d))
"My name is Joe."

>>> class PersonClass: pass
>>> p = PersonClass()
>>> p.first_name = "Ron"
>>> p.last_name = "Nasty"
>>> t.render(Context({"person": p}))
"My name is Ron."

>>> t = Template("The first stooge in the list is {{ stooges.0 }}.")
>>> c = Context({"stooges": ["Larry", "Curly", "Moe"]})
>>> t.render(c)
"The first stooge in the list is Larry."
```

If any part of the variable is callable, the template system will try calling it. Example:

```
>>> class PersonClass2:
...     def name(self):
...         return "Samantha"
>>> t = Template("My name is {{ person.name }}.")
>>> t.render(Context({"person": PersonClass2}))
"My name is Samantha."
```

Changed in version 1.3: Previously, only variables that originated with an attribute lookup would be called by the template system. This change was made for consistency across lookup types. Callable variables are slightly more complex than variables which only require straight lookups. Here are some things to keep in mind:

- If the variable raises an exception when called, the exception will be propagated, unless the exception has an attribute `silent_variable_failure` whose value is `True`. If the exception *does* have a `silent_variable_failure` attribute whose value is `True`, the variable will render as an empty string. Example:

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError("foo")
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(Exception):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

  Note that `django.core.exceptions.ObjectDoesNotExist`, which is the base class for all Django database API `DoesNotExist` exceptions, has `silent_variable_failure = True`. So if you're using Django templates with Django model objects, any `DoesNotExist` exception will fail silently.

- A variable can only be called if it has no required arguments. Otherwise, the system will return an empty string.

- Obviously, there can be side effects when calling some variables, and it'd be either foolish or a security hole to allow the template system to access them.

  A good example is the `delete()` method on each Django model object. The template system shouldn't be allowed to do something like this:

```
I will now delete this valuable data. {{ data.delete }}
```

  To prevent this, set an `alters_data` attribute on the callable variable. The template system won't call a variable if it has `alters_data=True` set, and will instead replace the variable with `TEMPLATE_STRING_IF_INVALID`, unconditionally. The dynamically-generated `delete()` and `save()` methods on Django model objects get `alters_data=True` automatically. Example:

```
def sensitive_function(self):
    self.database_record.delete()
sensitive_function.alters_data = True
```

- New in version 1.4: Occasionally you may want to turn off this feature for other reasons, and tell the template system to leave a variable un-called no matter what. To do so, set a `do_not_call_in_templates` attribute on the callable with the value `True`. The template system then will act as if your variable is not callable (allowing you to access attributes of the callable, for example).

How invalid variables are handled Generally, if a variable doesn't exist, the template system inserts the value of the `TEMPLATE_STRING_IF_INVALID` setting, which is set to `''` (the empty string) by default.

Filters that are applied to an invalid variable will only be applied if `TEMPLATE_STRING_IF_INVALID` is set to `''` (the empty string). If `TEMPLATE_STRING_IF_INVALID` is set to any other value, variable filters will be ignored.

This behavior is slightly different for the `if`, `for` and `regroup` template tags. If an invalid variable is provided to one of these template tags, the variable will be interpreted as `None`. Filters are always applied to invalid variables within these template tags.

If `TEMPLATE_STRING_IF_INVALID` contains a `'%s'`, the format marker will be replaced with the name of the invalid variable.

For debug purposes only!

While `TEMPLATE_STRING_IF_INVALID` can be a useful debugging tool, it is a bad idea to turn it on as a 'development default'.

Many templates, including those in the Admin site, rely upon the silence of the template system when a non-existent variable is encountered. If you assign a value other than `''` to `TEMPLATE_STRING_IF_INVALID`, you will experience rendering problems with these templates and sites.

Generally, `TEMPLATE_STRING_IF_INVALID` should only be enabled in order to debug a specific template problem, then cleared once debugging is complete.

Builtin variables Every context contains `True`, `False` and `None`. As you would expect, these variables resolve to the corresponding Python objects. New in version 1.5: Before Django 1.5, these variables weren't a special case, and they resolved to `None` unless you defined them in the context.

Playing with Context objects

class `django.template.`**`Context`**

Most of the time, you'll instantiate `Context` objects by passing in a fully-populated dictionary to `Context()`. But you can add and delete items from a `Context` object once it's been instantiated, too, using standard dictionary syntax:

```
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

`pop``()`

`push``()`

exception `django.template.`**`ContextPopException`**

A `Context` object is a stack. That is, you can `push()` and `pop()` it. If you `pop()` too much, it'll raise `django.template.ContextPopException`:

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.push()
>>> c['foo'] = 'second level'
>>> c['foo']
'second level'
>>> c.pop()
>>> c['foo']
```

```
'first level'
>>> c['foo'] = 'overwritten'
>>> c['foo']
'overwritten'
>>> c.pop()
Traceback (most recent call last):
...
django.template.ContextPopException
```

update(*other_dict*)

In addition to `push()` and `pop()`, the `Context` object also defines an `update()` method. This works like `push()` but takes a dictionary as an argument and pushes that dictionary onto the stack instead of an empty one.

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.update({'foo': 'updated'})
{'foo': 'updated'}
>>> c['foo']
'updated'
>>> c.pop()
{'foo': 'updated'}
>>> c['foo']
'first level'
```

Using a `Context` as a stack comes in handy in some custom template tags, as you'll see below.

Subclassing Context: RequestContext

class `django.template.`**RequestContext**

Django comes with a special `Context` class, `django.template.RequestContext`, that acts slightly differently than the normal `django.template.Context`. The first difference is that it takes an `HttpRequest` as its first argument. For example:

```
c = RequestContext(request, {
    'foo': 'bar',
})
```

The second difference is that it automatically populates the context with a few variables, according to your `TEMPLATE_CONTEXT_PROCESSORS` setting.

The `TEMPLATE_CONTEXT_PROCESSORS` setting is a tuple of callables – called **context processors** – that take a request object as their argument and return a dictionary of items to be merged into the context. By default, `TEMPLATE_CONTEXT_PROCESSORS` is set to:

```
("django.contrib.auth.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media",
"django.core.context_processors.static",
"django.core.context_processors.tz",
"django.contrib.messages.context_processors.messages")
```

In addition to these, `RequestContext` always uses `django.core.context_processors.csrf`. This is a security related context processor required by the admin and other contrib apps, and, in case of accidental misconfiguration, it is deliberately hardcoded in and cannot be turned off by the `TEMPLATE_CONTEXT_PROCESSORS` setting.

Each processor is applied in order. That means, if one processor adds a variable to the context and a second processor adds a variable with the same name, the second will override the first. The default processors are explained below.

When context processors are applied

When you use `RequestContext`, the variables you supply directly are added first, followed any variables supplied by context processors. This means that a context processor may overwrite a variable you've supplied, so take care to avoid variable names which overlap with those supplied by your context processors.

Also, you can give `RequestContext` a list of additional processors, using the optional, third positional argument, `processors`. In this example, the `RequestContext` instance gets a `ip_address` variable:

```
def ip_address_processor(request):
    return {'ip_address': request.META['REMOTE_ADDR']}

def some_view(request):
    # ...
    c = RequestContext(request, {
        'foo': 'bar',
    }, [ip_address_processor])
    return HttpResponse(t.render(c))
```

Note: If you're using Django's `render_to_response()` shortcut to populate a template with the contents of a dictionary, your template will be passed a `Context` instance by default (not a `RequestContext`). To use a `RequestContext` in your template rendering, pass an optional third argument to `render_to_response()`: a `RequestContext` instance. Your code might look like this:

```
def some_view(request):
    # ...
    return render_to_response('my_template.html',
                              my_data_dictionary,
                              context_instance=RequestContext(request))
```

Here's what each of the default processors does:

django.contrib.auth.context_processors.auth If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these three variables:

- `user` – An `auth.User` instance representing the currently logged-in user (or an `AnonymousUser` instance, if the client isn't logged in).
- `perms` – An instance of `django.contrib.auth.context_processors.PermWrapper`, representing the permissions that the currently logged-in user has.

Changed in version 1.3: Prior to version 1.3, `PermWrapper` was located in `django.contrib.auth.context_processors`.

django.core.context_processors.debug If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these two variables – but only if your `DEBUG` setting is set to `True` and the request's IP address (`request.META['REMOTE_ADDR']`) is in the `INTERNAL_IPS` setting:

- `debug` – `True`. You can use this in templates to test whether you're in `DEBUG` mode.
- `sql_queries` – A list of `{'sql': ..., 'time': ...}` dictionaries, representing every SQL query that has happened so far during the request and how long it took. The list is in order by query.

django.core.context_processors.i18n If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain these two variables:

- LANGUAGES – The value of the LANGUAGES setting.
- LANGUAGE_CODE – request.LANGUAGE_CODE, if it exists. Otherwise, the value of the LANGUAGE_CODE setting.

See *Internationalization and localization* for more.

django.core.context_processors.media If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a variable MEDIA_URL, providing the value of the MEDIA_URL setting.

django.core.context_processors.static
django.core.context_processors.**static**()
New in version 1.3: *Please see the release notes* If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a variable STATIC_URL, providing the value of the STATIC_URL setting.

django.core.context_processors.csrf This processor adds a token that is needed by the csrf_token template tag for protection against *Cross Site Request Forgeries*.

django.core.context_processors.request If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a variable request, which is the current HttpRequest. Note that this processor is not enabled by default; you'll have to activate it.

django.contrib.messages.context_processors.messages If TEMPLATE_CONTEXT_PROCESSORS contains this processor, every RequestContext will contain a single additional variable:

- messages – A list of messages (as strings) that have been set via the user model (using user.message_set.create) or through the *messages framework*.

Writing your own context processors A context processor has a very simple interface: It's just a Python function that takes one argument, an HttpRequest object, and returns a dictionary that gets added to the template context. Each context processor *must* return a dictionary.

Custom context processors can live anywhere in your code base. All Django cares about is that your custom context processors are pointed-to by your TEMPLATE_CONTEXT_PROCESSORS setting.

Loading templates

Generally, you'll store templates in files on your filesystem rather than using the low-level Template API yourself. Save templates in a directory specified as a **template directory**.

Django searches for template directories in a number of places, depending on your template-loader settings (see "Loader types" below), but the most basic way of specifying template directories is by using the TEMPLATE_DIRS setting.

The TEMPLATE_DIRS setting Tell Django what your template directories are by using the `TEMPLATE_DIRS` setting in your settings file. This should be set to a list or tuple of strings that contain full paths to your template directory(ies). Example:

```
TEMPLATE_DIRS = (
    "/home/html/templates/lawrence.com",
    "/home/html/templates/default",
)
```

Your templates can go anywhere you want, as long as the directories and templates are readable by the Web server. They can have any extension you want, such as `.html` or `.txt`, or they can have no extension at all.

Note that these paths should use Unix-style forward slashes, even on Windows.

The Python API Django has two ways to load templates from files:

`django.template.loader.`**`get_template`**`(`*template_name*`)`
`get_template` returns the compiled template (a `Template` object) for the template with the given name. If the template doesn't exist, it raises `django.template.TemplateDoesNotExist`.

`django.template.loader.`**`select_template`**`(`*template_name_list*`)`
`select_template` is just like `get_template`, except it takes a list of template names. Of the list, it returns the first template that exists.

For example, if you call `get_template('story_detail.html')` and have the above `TEMPLATE_DIRS` setting, here are the files Django will look for, in order:

- `/home/html/templates/lawrence.com/story_detail.html`
- `/home/html/templates/default/story_detail.html`

If you call `select_template(['story_253_detail.html', 'story_detail.html'])`, here's what Django will look for:

- `/home/html/templates/lawrence.com/story_253_detail.html`
- `/home/html/templates/default/story_253_detail.html`
- `/home/html/templates/lawrence.com/story_detail.html`
- `/home/html/templates/default/story_detail.html`

When Django finds a template that exists, it stops looking.

Tip

You can use `select_template()` for super-flexible "templatability." For example, if you've written a news story and want some stories to have custom templates, use something like `select_template(['story_%s_detail.html' % story.id, 'story_detail.html'])`. That'll allow you to use a custom template for an individual story, with a fallback template for stories that don't have custom templates.

Using subdirectories It's possible – and preferable – to organize templates in subdirectories of the template directory. The convention is to make a subdirectory for each Django app, with subdirectories within those subdirectories as needed.

Do this for your own sanity. Storing all templates in the root level of a single directory gets messy.

To load a template that's within a subdirectory, just use a slash, like so:

```
get_template('news/story_detail.html')
```

Using the same TEMPLATE_DIRS setting from above, this example get_template() call will attempt to load the following templates:

- /home/html/templates/lawrence.com/news/story_detail.html
- /home/html/templates/default/news/story_detail.html

Loader types By default, Django uses a filesystem-based template loader, but Django comes with a few other template loaders, which know how to load templates from other sources.

Some of these other loaders are disabled by default, but you can activate them by editing your TEMPLATE_LOADERS setting. TEMPLATE_LOADERS should be a tuple of strings, where each string represents a template loader class. Here are the template loaders that come with Django:

django.template.loaders.filesystem.Loader Loads templates from the filesystem, according to TEMPLATE_DIRS. This loader is enabled by default.

django.template.loaders.app_directories.Loader Loads templates from Django apps on the filesystem. For each app in INSTALLED_APPS, the loader looks for a templates subdirectory. If the directory exists, Django looks for templates in there.

This means you can store templates with your individual apps. This also makes it easy to distribute Django apps with default templates.

For example, for this setting:

```
INSTALLED_APPS = ('myproject.polls', 'myproject.music')
```

...then get_template('foo.html') will look for foo.html in these directories, in this order:

- /path/to/myproject/polls/templates/
- /path/to/myproject/music/templates/

... and will use the one it finds first.

The order of INSTALLED_APPS is significant! For example, if you want to customize the Django admin, you might choose to override the standard admin/base_site.html template, from django.contrib.admin, with your own admin/base_site.html in myproject.polls. You must then make sure that your myproject.polls comes *before* django.contrib.admin in INSTALLED_APPS, otherwise django.contrib.admin's will be loaded first and yours will be ignored.

Note that the loader performs an optimization when it is first imported: it caches a list of which INSTALLED_APPS packages have a templates subdirectory.

This loader is enabled by default.

django.template.loaders.eggs.Loader Just like app_directories above, but it loads templates from Python eggs rather than from the filesystem.

This loader is disabled by default.

django.template.loaders.cached.Loader By default, the templating system will read and compile your templates every time they need to be rendered. While the Django templating system is quite fast, the overhead from reading and compiling templates can add up.

The cached template loader is a class-based loader that you configure with a list of other loaders that it should wrap. The wrapped loaders are used to locate unknown templates when they are first encountered. The cached loader then stores the compiled Template in memory. The cached Template instance is returned for subsequent requests to load the same template.

For example, to enable template caching with the `filesystem` and `app_directories` template loaders you might use the following settings:

```
TEMPLATE_LOADERS = (
    ('django.template.loaders.cached.Loader', (
        'django.template.loaders.filesystem.Loader',
        'django.template.loaders.app_directories.Loader',
    )),
)
```

Note: All of the built-in Django template tags are safe to use with the cached loader, but if you're using custom template tags that come from third party packages, or that you wrote yourself, you should ensure that the `Node` implementation for each tag is thread-safe. For more information, see *template tag thread safety considerations*.

This loader is disabled by default.

Django uses the template loaders in order according to the `TEMPLATE_LOADERS` setting. It uses each loader until a loader finds a match.

The `render_to_string` shortcut

`django.template.loader.`**`render_to_string`**(*template_name*, *dictionary=None*, *context_instance=None*)

To cut down on the repetitive nature of loading and rendering templates, Django provides a shortcut function which largely automates the process: `render_to_string()` in `django.template.loader`, which loads a template, renders it and returns the resulting string:

```
from django.template.loader import render_to_string
rendered = render_to_string('my_template.html', {'foo': 'bar'})
```

The `render_to_string` shortcut takes one required argument – `template_name`, which should be the name of the template to load and render (or a list of template names, in which case Django will use the first template in the list that exists) – and two optional arguments:

dictionary A dictionary to be used as variables and values for the template's context. This can also be passed as the second positional argument.

context_instance An instance of `Context` or a subclass (e.g., an instance of `RequestContext`) to use as the template's context. This can also be passed as the third positional argument.

See also the `render_to_response()` shortcut, which calls `render_to_string` and feeds the result into an `HttpResponse` suitable for returning directly from a view.

Configuring the template system in standalone mode

Note: This section is only of interest to people trying to use the template system as an output component in another application. If you're using the template system as part of a Django application, nothing here applies to you.

Normally, Django will load all the configuration information it needs from its own default configuration file, combined with the settings in the module given in the `DJANGO_SETTINGS_MODULE` environment variable. But if you're using the template system independently of the rest of Django, the environment variable approach isn't very convenient, because you probably want to configure the template system in line with the rest of your application rather than dealing with settings files and pointing to them via environment variables.

To solve this problem, you need to use the manual configuration option described in *Using settings without setting DJANGO_SETTINGS_MODULE*. Simply import the appropriate pieces of the templating system and then, *before* you call any of the templating functions, call `django.conf.settings.configure()` with any settings you wish to specify. You might want to consider setting at least `TEMPLATE_DIRS` (if you're going to use template loaders), `DEFAULT_CHARSET` (although the default of `utf-8` is probably fine) and `TEMPLATE_DEBUG`. If you plan to use the `url` template tag, you will also need to set the `ROOT_URLCONF` setting. All available settings are described in the *settings documentation*, and any setting starting with `TEMPLATE_` is of obvious interest.

Using an alternative template language

The Django `Template` and `Loader` classes implement a simple API for loading and rendering templates. By providing some simple wrapper classes that implement this API we can use third party template systems like Jinja2 or Cheetah. This allows us to use third-party template libraries without giving up useful Django features like the Django `Context` object and handy shortcuts like `render_to_response()`.

The core component of the Django templating system is the `Template` class. This class has a very simple interface: it has a constructor that takes a single positional argument specifying the template string, and a `render()` method that takes a `Context` object and returns a string containing the rendered response.

Suppose we're using a template language that defines a `Template` object with a `render()` method that takes a dictionary rather than a `Context` object. We can write a simple wrapper that implements the Django `Template` interface:

```
import some_template_language
class Template(some_template_language.Template):
    def render(self, context):
        # flatten the Django Context into a single dictionary.
        context_dict = {}
        for d in context.dicts:
            context_dict.update(d)
        return super(Template, self).render(context_dict)
```

That's all that's required to make our fictional `Template` class compatible with the Django loading and rendering system!

The next step is to write a `Loader` class that returns instances of our custom template class instead of the default `Template`. Custom `Loader` classes should inherit from `django.template.loader.BaseLoader` and override the `load_template_source()` method, which takes a `template_name` argument, loads the template from disk (or elsewhere), and returns a tuple: `(template_string, template_origin)`.

The `load_template()` method of the `Loader` class retrieves the template string by calling `load_template_source()`, instantiates a `Template` from the template source, and returns a tuple: `(template, template_origin)`. Since this is the method that actually instantiates the `Template`, we'll need to override it to use our custom template class instead. We can inherit from the builtin `django.template.loaders.app_directories.Loader` to take advantage of the `load_template_source()` method implemented there:

```
from django.template.loaders import app_directories
class Loader(app_directories.Loader):
    is_usable = True

    def load_template(self, template_name, template_dirs=None):
        source, origin = self.load_template_source(template_name, template_dirs)
        template = Template(source)
        return template, origin
```

Finally, we need to modify our project settings, telling Django to use our custom loader. Now we can write all of our templates in our alternative template language while continuing to use the rest of the Django templating system.

See Also:

For information on writing your own custom tags and filters, see *Custom template tags and filters*.

6.18 Unicode data

Django natively supports Unicode data everywhere. Providing your database can somehow store the data, you can safely pass around Unicode strings to templates, models and the database.

This document tells you what you need to know if you're writing applications that use data or templates that are encoded in something other than ASCII.

6.18.1 Creating the database

Make sure your database is configured to be able to store arbitrary string data. Normally, this means giving it an encoding of UTF-8 or UTF-16. If you use a more restrictive encoding – for example, latin1 (iso8859-1) – you won't be able to store certain characters in the database, and information will be lost.

- MySQL users, refer to the MySQL manual (section 9.1.3.2 for MySQL 5.1) for details on how to set or alter the database character set encoding.
- PostgreSQL users, refer to the PostgreSQL manual (section 21.2.2 in PostgreSQL 8) for details on creating databases with the correct encoding.
- SQLite users, there is nothing you need to do. SQLite always uses UTF-8 for internal encoding.

All of Django's database backends automatically convert Unicode strings into the appropriate encoding for talking to the database. They also automatically convert strings retrieved from the database into Python Unicode strings. You don't even need to tell Django what encoding your database uses: that is handled transparently.

For more, see the section "The database API" below.

6.18.2 General string handling

Whenever you use strings with Django – e.g., in database lookups, template rendering or anywhere else – you have two choices for encoding those strings. You can use Unicode strings, or you can use normal strings (sometimes called "bytestrings") that are encoded using UTF-8. Changed in version 1.5: *Please see the release notes* In Python 3, the logic is reversed, that is normal strings are Unicode, and when you want to specifically create a bytestring, you have to prefix the string with a 'b'. As we are doing in Django code from version 1.5, we recommend that you import `unicode_literals` from the __future__ library in your code. Then, when you specifically want to create a bytestring literal, prefix the string with 'b'.

Python 2 legacy:

```
my_string = "This is a bytestring"
my_unicode = u"This is an Unicode string"
```

Python 2 with unicode literals or Python 3:

```
from __future__ import unicode_literals

my_string = b"This is a bytestring"
my_unicode = "This is an Unicode string"
```

See also *Python 3 compatibility*.

Warning

A bytestring does not carry any information with it about its encoding. For that reason, we have to make an assumption, and Django assumes that all bytestrings are in UTF-8.

If you pass a string to Django that has been encoded in some other format, things will go wrong in interesting ways. Usually, Django will raise a `UnicodeDecodeError` at some point.

If your code only uses ASCII data, it's safe to use your normal strings, passing them around at will, because ASCII is a subset of UTF-8.

Don't be fooled into thinking that if your `DEFAULT_CHARSET` setting is set to something other than `'utf-8'` you can use that other encoding in your bytestrings! `DEFAULT_CHARSET` only applies to the strings generated as the result of template rendering (and email). Django will always assume UTF-8 encoding for internal bytestrings. The reason for this is that the `DEFAULT_CHARSET` setting is not actually under your control (if you are the application developer). It's under the control of the person installing and using your application – and if that person chooses a different setting, your code must still continue to work. Ergo, it cannot rely on that setting.

In most cases when Django is dealing with strings, it will convert them to Unicode strings before doing anything else. So, as a general rule, if you pass in a bytestring, be prepared to receive a Unicode string back in the result.

Translated strings

Aside from Unicode strings and bytestrings, there's a third type of string-like object you may encounter when using Django. The framework's internationalization features introduce the concept of a "lazy translation" – a string that has been marked as translated but whose actual translation result isn't determined until the object is used in a string. This feature is useful in cases where the translation locale is unknown until the string is used, even though the string might have originally been created when the code was first imported.

Normally, you won't have to worry about lazy translations. Just be aware that if you examine an object and it claims to be a `django.utils.functional.__proxy__` object, it is a lazy translation. Calling `unicode()` with the lazy translation as the argument will generate a Unicode string in the current locale.

For more details about lazy translation objects, refer to the *internationalization* documentation.

Useful utility functions

Because some string operations come up again and again, Django ships with a few useful functions that should make working with Unicode and bytestring objects a bit easier.

Conversion functions

The `django.utils.encoding` module contains a few functions that are handy for converting back and forth between Unicode and bytestrings.

- `smart_unicode(s, encoding='utf-8', strings_only=False, errors='strict')` converts its input to a Unicode string. The `encoding` parameter specifies the input encoding. (For example, Django uses this internally when processing form input data, which might not be UTF-8 encoded.) The `strings_only` parameter, if set to True, will result in Python numbers, booleans and `None` not being converted to a string (they keep their original types). The `errors` parameter takes any of the values that are accepted by Python's `unicode()` function for its error handling.

If you pass `smart_unicode()` an object that has a `__unicode__` method, it will use that method to do the conversion.

- `force_unicode(s, encoding='utf-8', strings_only=False, errors='strict')` is identical to `smart_unicode()` in almost all cases. The difference is when the first argument is a *lazy translation* instance. While `smart_unicode()` preserves lazy translations, `force_unicode()` forces those objects to a Unicode string (causing the translation to occur). Normally, you'll want to use `smart_unicode()`. However, `force_unicode()` is useful in template tags and filters that absolutely *must* have a string to work with, not just something that can be converted to a string.
- `smart_str(s, encoding='utf-8', strings_only=False, errors='strict')` is essentially the opposite of `smart_unicode()`. It forces the first argument to a bytestring. The `strings_only` parameter has the same behavior as for `smart_unicode()` and `force_unicode()`. This is slightly different semantics from Python's builtin `str()` function, but the difference is needed in a few places within Django's internals.

Normally, you'll only need to use `smart_unicode()`. Call it as early as possible on any input data that might be either Unicode or a bytestring, and from then on, you can treat the result as always being Unicode.

URI and IRI handling

Web frameworks have to deal with URLs (which are a type of IRI). One requirement of URLs is that they are encoded using only ASCII characters. However, in an international environment, you might need to construct a URL from an IRI – very loosely speaking, a URI that can contain Unicode characters. Quoting and converting an IRI to URI can be a little tricky, so Django provides some assistance.

- The function `django.utils.encoding.iri_to_uri()` implements the conversion from IRI to URI as required by the specification (**RFC 3987**).
- The functions `django.utils.http.urlquote()` and `django.utils.http.urlquote_plus()` are versions of Python's standard `urllib.quote()` and `urllib.quote_plus()` that work with non-ASCII characters. (The data is converted to UTF-8 prior to encoding.)

These two groups of functions have slightly different purposes, and it's important to keep them straight. Normally, you would use `urlquote()` on the individual portions of the IRI or URI path so that any reserved characters such as '&' or '%' are correctly encoded. Then, you apply `iri_to_uri()` to the full IRI and it converts any non-ASCII characters to the correct encoded values.

Note: Technically, it isn't correct to say that `iri_to_uri()` implements the full algorithm in the IRI specification. It doesn't (yet) perform the international domain name encoding portion of the algorithm.

The `iri_to_uri()` function will not change ASCII characters that are otherwise permitted in a URL. So, for example, the character '%' is not further encoded when passed to `iri_to_uri()`. This means you can pass a full URL to this function and it will not mess up the query string or anything like that.

An example might clarify things here:

```
>>> urlquote(u'Paris & Orléans')
u'Paris%20%26%20Orl%C3%A9ans'
>>> iri_to_uri(u'/favorites/François/%s' % urlquote('Paris & Orléans'))
'/favorites/Fran%C3%A7ois/Paris%20%26%20Orl%C3%A9ans'
```

If you look carefully, you can see that the portion that was generated by `urlquote()` in the second example was not double-quoted when passed to `iri_to_uri()`. This is a very important and useful feature. It means that you can construct your IRI without worrying about whether it contains non-ASCII characters and then, right at the end, call `iri_to_uri()` on the result.

The `iri_to_uri()` function is also idempotent, which means the following is always true:

```
iri_to_uri(iri_to_uri(some_string)) = iri_to_uri(some_string)
```

So you can safely call it multiple times on the same IRI without risking double-quoting problems.

6.18.3 Models

Because all strings are returned from the database as Unicode strings, model fields that are character based (CharField, TextField, URLField, etc) will contain Unicode values when Django retrieves data from the database. This is *always* the case, even if the data could fit into an ASCII bytestring.

You can pass in bytestrings when creating a model or populating a field, and Django will convert it to Unicode when it needs to.

Choosing between `__str__()` and `__unicode__()`

One consequence of using Unicode by default is that you have to take some care when printing data from the model.

In particular, rather than giving your model a `__str__()` method, we recommended you implement a `__unicode__()` method. In the `__unicode__()` method, you can quite safely return the values of all your fields without having to worry about whether they fit into a bytestring or not. (The way Python works, the result of `__str__()` is *always* a bytestring, even if you accidentally try to return a Unicode object).

You can still create a `__str__()` method on your models if you want, of course, but you shouldn't need to do this unless you have a good reason. Django's `Model` base class automatically provides a `__str__()` implementation that calls `__unicode__()` and encodes the result into UTF-8. This means you'll normally only need to implement a `__unicode__()` method and let Django handle the coercion to a bytestring when required.

Taking care in `get_absolute_url()`

URLs can only contain ASCII characters. If you're constructing a URL from pieces of data that might be non-ASCII, be careful to encode the results in a way that is suitable for a URL. The `django.db.models.permalink()` decorator handles this for you automatically.

If you're constructing a URL manually (i.e., *not* using the `permalink()` decorator), you'll need to take care of the encoding yourself. In this case, use the `iri_to_uri()` and `urlquote()` functions that were documented above. For example:

```
from django.utils.encoding import iri_to_uri
from django.utils.http import urlquote

def get_absolute_url(self):
    url = u'/person/%s/?x=0&y=0' % urlquote(self.location)
    return iri_to_uri(url)
```

This function returns a correctly encoded URL even if `self.location` is something like "Jack visited Paris & Orléans". (In fact, the `iri_to_uri()` call isn't strictly necessary in the above example, because all the non-ASCII characters would have been removed in quoting in the first line.)

6.18.4 The database API

You can pass either Unicode strings or UTF-8 bytestrings as arguments to `filter()` methods and the like in the database API. The following two querysets are identical:

```
from __future__ import unicode_literals

qs = People.objects.filter(name__contains='Å')
qs = People.objects.filter(name__contains=b'\xc3\x85') # UTF-8 encoding of Å
```

6.18.5 Templates

You can use either Unicode or bytestrings when creating templates manually:

```
from __future__ import unicode_literals
from django.template import Template
t1 = Template(b'This is a bytestring template.')
t2 = Template('This is a Unicode template.')
```

But the common case is to read templates from the filesystem, and this creates a slight complication: not all filesystems store their data encoded as UTF-8. If your template files are not stored with a UTF-8 encoding, set the `FILE_CHARSET` setting to the encoding of the files on disk. When Django reads in a template file, it will convert the data from this encoding to Unicode. (`FILE_CHARSET` is set to `'utf-8'` by default.)

The `DEFAULT_CHARSET` setting controls the encoding of rendered templates. This is set to UTF-8 by default.

Template tags and filters

A couple of tips to remember when writing your own template tags and filters:

- Always return Unicode strings from a template tag's `render()` method and from template filters.
- Use `force_unicode()` in preference to `smart_unicode()` in these places. Tag rendering and filter calls occur as the template is being rendered, so there is no advantage to postponing the conversion of lazy translation objects into strings. It's easier to work solely with Unicode strings at that point.

6.18.6 Email

Django's email framework (in `django.core.mail`) supports Unicode transparently. You can use Unicode data in the message bodies and any headers. However, you're still obligated to respect the requirements of the email specifications, so, for example, email addresses should use only ASCII characters.

The following code example demonstrates that everything except email addresses can be non-ASCII:

```
from __future__ import unicode_literals
from django.core.mail import EmailMessage

subject = 'My visit to Sør-Trøndelag'
sender = 'Arnbjörg Ráðormsdóttir <arnbjorg@example.com>'
recipients = ['Fred <fred@example.com']
body = '...'
msg = EmailMessage(subject, body, sender, recipients)
msg.attach("Une pièce jointe.pdf", "%PDF-1.4.%...", mimetype="application/pdf")
msg.send()
```

6.18.7 Form submission

HTML form submission is a tricky area. There's no guarantee that the submission will include encoding information, which means the framework might have to guess at the encoding of submitted data.

Django adopts a "lazy" approach to decoding form data. The data in an `HttpRequest` object is only decoded when you access it. In fact, most of the data is not decoded at all. Only the `HttpRequest.GET` and `HttpRequest.POST` data structures have any decoding applied to them. Those two fields will return their members as Unicode data. All other attributes and methods of `HttpRequest` return data exactly as it was submitted by the client.

By default, the `DEFAULT_CHARSET` setting is used as the assumed encoding for form data. If you need to change this for a particular form, you can set the `encoding` attribute on an `HttpRequest` instance. For example:

```
def some_view(request):
    # We know that the data must be encoded as KOI8-R (for some reason).
    request.encoding = 'koi8-r'
    ...
```

You can even change the encoding after having accessed `request.GET` or `request.POST`, and all subsequent accesses will use the new encoding.

Most developers won't need to worry about changing form encoding, but this is a useful feature for applications that talk to legacy systems whose encoding you cannot control.

Django does not decode the data of file uploads, because that data is normally treated as collections of bytes, rather than strings. Any automatic decoding there would alter the meaning of the stream of bytes.

6.19 Django Utils

This document covers all stable modules in `django.utils`. Most of the modules in `django.utils` are designed for internal use and only the following parts can be considered stable and thus backwards compatible as per the *internal release deprecation policy*.

6.19.1 `django.utils.cache`

This module contains helper functions for controlling caching. It does so by managing the `Vary` header of responses. It includes functions to patch the header of response objects directly and decorators that change functions to do that header-patching themselves.

For information on the `Vary` header, see **RFC 2616** section 14.44.

Essentially, the `Vary` HTTP header defines which headers a cache should take into account when building its cache key. Requests with the same path but different header content for headers named in `Vary` need to get different cache keys to prevent delivery of wrong content.

For example, *internationalization* middleware would need to distinguish caches by the `Accept-language` header.

patch_cache_control(*response*, ***kwargs*)

This function patches the `Cache-Control` header by adding all keyword arguments to it. The transformation is as follows:

- All keyword parameter names are turned to lowercase, and underscores are converted to hyphens.
- If the value of a parameter is `True` (exactly `True`, not just a true value), only the parameter name is added to the header.
- All other parameters are added with their value, after applying `str()` to it.

get_max_age(*response*)

Returns the max-age from the response Cache-Control header as an integer (or `None` if it wasn't found or wasn't an integer).

patch_response_headers(*response*, *cache_timeout=None*)
Adds some useful headers to the given `HttpResponse` object:

- `ETag`
- `Last-Modified`
- `Expires`
- `Cache-Control`

Each header is only added if it isn't already set.

`cache_timeout` is in seconds. The `CACHE_MIDDLEWARE_SECONDS` setting is used by default.

add_never_cache_headers(*response*)
Adds headers to a response to indicate that a page should never be cached.

patch_vary_headers(*response*, *newheaders*)
Adds (or updates) the `Vary` header in the given `HttpResponse` object. `newheaders` is a list of header names that should be in `Vary`. Existing headers in `Vary` aren't removed.

get_cache_key(*request*, *key_prefix=None*)
Returns a cache key based on the request path. It can be used in the request phase because it pulls the list of headers to take into account from the global path registry and uses those to build a cache key to check against.

If there is no headerlist stored, the page needs to be rebuilt, so this function returns `None`.

learn_cache_key(*request*, *response*, *cache_timeout=None*, *key_prefix=None*)
Learns what headers to take into account for some request path from the response object. It stores those headers in a global path registry so that later access to that path will know what headers to take into account without building the response object itself. The headers are named in the `Vary` header of the response, but we want to prevent response generation.

The list of headers to use for cache key generation is stored in the same cache as the pages themselves. If the cache ages some data out of the cache, this just means that we have to build the response once to get at the Vary header and so at the list of headers to use for the cache key.

6.19.2 `django.utils.datastructures`

class SortedDict
The `django.utils.datastructures.SortedDict` class is a dictionary that keeps its keys in the order in which they're inserted. `SortedDict` adds two additional methods to the standard Python `dict` class:

insert(*index*, *key*, *value*)
Deprecated since version 1.5. Inserts the key, value pair before the item with the given index.

value_for_index(*index*)
Deprecated since version 1.5. Returns the value of the item at the given zero-based index.

Creating a new SortedDict

Creating a new `SortedDict` must be done in a way where ordering is guaranteed. For example:

```
SortedDict({'b': 1, 'a': 2, 'c': 3})
```

will not work. Passing in a basic Python `dict` could produce unreliable results. Instead do:

```
SortedDict([('b', 1), ('a', 2), ('c', 3)])
```

6.19.3 django.utils.dateparse

New in version 1.4: *Please see the release notes* The functions defined in this module share the following properties:

- They raise `ValueError` if their input is well formatted but isn't a valid date or time.
- They return `None` if it isn't well formatted at all.
- They accept up to picosecond resolution in input, but they truncate it to microseconds, since that's what Python supports.

parse_date(*value*)
Parses a string and returns a `datetime.date`.

parse_time(*value*)
Parses a string and returns a `datetime.time`.

UTC offsets aren't supported; if `value` describes one, the result is `None`.

parse_datetime(*value*)
Parses a string and returns a `datetime.datetime`.

UTC offsets are supported; if `value` describes one, the result's `tzinfo` attribute is a `FixedOffset` instance.

6.19.4 django.utils.encoding

class StrAndUnicode
A class whose `__str__` returns its `__unicode__` as a UTF-8 bytestring. Useful as a mix-in.

smart_unicode(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Returns a `unicode` object representing `s`. Treats bytestrings using the 'encoding' codec.

If `strings_only` is `True`, don't convert (some) non-string-like objects.

is_protected_type(*obj*)
Determine if the object instance is of a protected type.

Objects of protected types are preserved as-is when passed to `force_unicode(strings_only=True)`.

force_unicode(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Similar to `smart_unicode`, except that lazy instances are resolved to strings, rather than kept as lazy objects.

If `strings_only` is `True`, don't convert (some) non-string-like objects.

smart_str(*s*, *encoding='utf-8'*, *strings_only=False*, *errors='strict'*)
Returns a bytestring version of `s`, encoded as specified in `encoding`.

If `strings_only` is `True`, don't convert (some) non-string-like objects.

iri_to_uri(*iri*)
Convert an Internationalized Resource Identifier (IRI) portion to a URI portion that is suitable for inclusion in a URL.

This is the algorithm from section 3.1 of **RFC 3987**. However, since we are assuming input is either UTF-8 or unicode already, we can simplify things a little from the full method.

Returns an ASCII string containing the encoded result.

6.19.5 django.utils.feedgenerator

Sample usage:

```
>>> from django.utils import feedgenerator
>>> feed = feedgenerator.Rss201rev2Feed(
...     title=u"Poynter E-Media Tidbits",
...     link=u"http://www.poynter.org/column.asp?id=31",
...     description=u"A group Weblog by the sharpest minds in online media/journalism/publishing.",
...     language=u"en",
... )
>>> feed.add_item(
...     title="Hello",
...     link=u"http://www.holovaty.com/test/",
...     description="Testing."
... )
>>> with open('test.rss', 'w') as fp:
...     feed.write(fp, 'utf-8')
```

For simplifying the selection of a generator use `feedgenerator.DefaultFeed` which is currently `Rss201rev2Feed`

For definitions of the different versions of RSS, see: http://web.archive.org/web/20110718035220/http://diveintomark.org/archives/2004/rss

get_tag_uri(*url*, *date*)
Creates a TagURI.

See http://web.archive.org/web/20110514113830/http://diveintomark.org/archives/2004/05/28/howto-atom-id

SyndicationFeed

class SyndicationFeed
Base class for all syndication feeds. Subclasses should provide write().

__init__(*title*, *link*, *description*[, *language=None*, *author_email=None*, *author_name=None*, *author_link=None*, *subtitle=None*, *categories=None*, *feed_url=None*, *feed_copyright=None*, *feed_guid=None*, *ttl=None*, ***kwargs*])
Initialize the feed with the given dictionary of metadata, which applies to the entire feed.

Any extra keyword arguments you pass to `__init__` will be stored in `self.feed`.

All parameters should be Unicode objects, except `categories`, which should be a sequence of Unicode objects.

add_item(*title*, *link*, *description*[, *author_email=None*, *author_name=None*, *author_link=None*, *pubdate=None*, *comments=None*, *unique_id=None*, *enclosure=None*, *categories=()*, *item_copyright=None*, *ttl=None*, ***kwargs*])
Adds an item to the feed. All args are expected to be Python `unicode` objects except `pubdate`, which is a `datetime.datetime` object, and `enclosure`, which is an instance of the `Enclosure` class.

num_items()

root_attributes()
Return extra attributes to place on the root (i.e. feed/channel) element. Called from `write()`.

add_root_elements(*handler*)
Add elements in the root (i.e. feed/channel) element. Called from `write()`.

item_attributes(*item*)
Return extra attributes to place on each item (i.e. item/entry) element.

add_item_elements(*handler*, *item*)
Add elements on each item (i.e. item/entry) element.

write(*outfile*, *encoding*)
Outputs the feed in the given encoding to `outfile`, which is a file-like object. Subclasses should override this.

writeString(*encoding*)
Returns the feed in the given encoding as a string.

latest_post_date()
Returns the latest item's `pubdate`. If none of them have a `pubdate`, this returns the current date/time.

Enclosure

class Enclosure
Represents an RSS enclosure

RssFeed

class RssFeed(*SyndicationFeed*)

Rss201rev2Feed

class Rss201rev2Feed(*RssFeed*)
Spec: http://cyber.law.harvard.edu/rss/rss.html

RssUserland091Feed

class RssUserland091Feed(*RssFeed*)
Spec: http://backend.userland.com/rss091

Atom1Feed

class Atom1Feed(*SyndicationFeed*)
Spec: http://www.atomenabled.org/developers/syndication/atom-format-spec.php

6.19.6 `django.utils.functional`

allow_lazy(*func*, **resultclasses*)
Django offers many utility functions (particularly in `django.utils`) that take a string as their first argument and do something to that string. These functions are used by template filters as well as directly in other code.

If you write your own similar functions and deal with translations, you'll face the problem of what to do when the first argument is a lazy translation object. You don't want to convert it to a string immediately, because you might be using this function outside of a view (and hence the current thread's locale setting will not be correct).

For cases like this, use the `django.utils.functional.allow_lazy()` decorator. It modifies the function so that *if* it's called with a lazy translation as the first argument, the function evaluation is delayed until it needs to be converted to a string.

For example:

```
from django.utils.functional import allow_lazy

def fancy_utility_function(s, ...):
    # Do some conversion on string 's'
    ...
fancy_utility_function = allow_lazy(fancy_utility_function, unicode)
```

The `allow_lazy()` decorator takes, in addition to the function to decorate, a number of extra arguments (`*args`) specifying the type(s) that the original function can return. Usually, it's enough to include `unicode` here and ensure that your function returns only Unicode strings.

Using this decorator means you can write your function and assume that the input is a proper string, then add support for lazy translation objects at the end.

6.19.7 django.utils.html

Usually you should build up HTML using Django's templates to make use of its autoescape mechanism, using the utilities in `django.utils.safestring` where appropriate. This module provides some additional low level utilitiesfor escaping HTML.

escape(*text*)
Returns the given text with ampersands, quotes and angle brackets encoded for use in HTML. The input is first passed through `force_unicode()` and the output has `mark_safe()` applied.

conditional_escape(*text*)
Similar to `escape()`, except that it doesn't operate on pre-escaped strings, so it will not double escape.

format_html(*format_string*, **args*, ***kwargs*)
This is similar to str.format, except that it is appropriate for building up HTML fragments. All args and kwargs are passed through `conditional_escape()` before being passed to `str.format`.

For the case of building up small HTML fragments, this function is to be preferred over string interpolation using `%` or `str.format` directly, because it applies escaping to all arguments - just like the Template system applies escaping by default.

So, instead of writing:

```
mark_safe(u"%s <b>%s</b> %s" % (some_html,
                                escape(some_text),
                                escape(some_other_text),
                                ))
```

you should instead use:

```
format_html(u"{0} <b>{1}</b> {2}",
            mark_safe(some_html), some_text, some_other_text)
```

This has the advantage that you don't need to apply `escape()` to each argument and risk a bug and an XSS vulnerability if you forget one.

Note that although this function uses `str.format` to do the interpolation, some of the formatting options provided by str.format (e.g. number formatting) will not work, since all arguments are passed through `conditional_escape()` which (ultimately) calls `force_unicode()` on the values.

6.19.8 django.utils.http

urlquote(*url*, *safe='/'*)
A version of Python's `urllib.quote()` function that can operate on unicode strings. The url is first UTF-

8 encoded before quoting. The returned string can safely be used as part of an argument to a subsequent `iri_to_uri()` call without double-quoting occurring. Employs lazy execution.

urlquote_plus(*url*, *safe=''*)
A version of Python's urllib.quote_plus() function that can operate on unicode strings. The url is first UTF-8 encoded before quoting. The returned string can safely be used as part of an argument to a subsequent `iri_to_uri()` call without double-quoting occurring. Employs lazy execution.

urlencode(*query*, *doseq=0*)
A version of Python's urllib.urlencode() function that can operate on unicode strings. The parameters are first case to UTF-8 encoded strings and then encoded as per normal.

cookie_date(*epoch_seconds=None*)
Formats the time to ensure compatibility with Netscape's cookie standard.

Accepts a floating point number expressed in seconds since the epoch in UTC–such as that outputted by `time.time()`. If set to `None`, defaults to the current time.

Outputs a string in the format `Wdy, DD-Mon-YYYY HH:MM:SS GMT`.

http_date(*epoch_seconds=None*)
Formats the time to match the **RFC 1123** date format as specified by HTTP **RFC 2616** section 3.3.1.

Accepts a floating point number expressed in seconds since the epoch in UTC–such as that outputted by `time.time()`. If set to `None`, defaults to the current time.

Outputs a string in the format `Wdy, DD Mon YYYY HH:MM:SS GMT`.

base36_to_int(*s*)
Converts a base 36 string to an integer.

int_to_base36(*i*)
Converts a positive integer less than sys.maxint to a base 36 string.

6.19.9 `django.utils.safestring`

Functions and classes for working with "safe strings": strings that can be displayed safely without further escaping in HTML. Marking something as a "safe string" means that the producer of the string has already turned characters that should not be interpreted by the HTML engine (e.g. '<') into the appropriate entities.

class SafeString
A string subclass that has been specifically marked as "safe" (requires no further escaping) for HTML output purposes.

class SafeUnicode
A unicode subclass that has been specifically marked as "safe" for HTML output purposes.

mark_safe(*s*)
Explicitly mark a string as safe for (HTML) output purposes. The returned object can be used everywhere a string or unicode object is appropriate.

Can be called multiple times on a single string.

mark_for_escaping(*s*)
Explicitly mark a string as requiring HTML escaping upon output. Has no effect on `SafeData` subclasses.

Can be called multiple times on a single string (the resulting escaping is only applied once).

6.19.10 django.utils.translation

For a complete discussion on the usage of the following see the *translation documentation*.

gettext(*message*)
Translates `message` and returns it in a UTF-8 bytestring

ugettext(*message*)
Translates `message` and returns it in a unicode string

pgettext(*context*, *message*)
Translates `message` given the `context` and returns it in a unicode string.

For more information, see *Contextual markers*.

gettext_lazy(*message*)

ugettext_lazy(*message*)

pgettext_lazy(*context*, *message*)
Same as the non-lazy versions above, but using lazy execution.

See *lazy translations documentation*.

gettext_noop(*message*)

ugettext_noop(*message*)
Marks strings for translation but doesn't translate them now. This can be used to store strings in global variables that should stay in the base language (because they might be used externally) and will be translated later.

ngettext(*singular*, *plural*, *number*)
Translates `singular` and `plural` and returns the appropriate string based on `number` in a UTF-8 bytestring.

ungettext(*singular*, *plural*, *number*)
Translates `singular` and `plural` and returns the appropriate string based on `number` in a unicode string.

npgettext(*context*, *singular*, *plural*, *number*)
Translates `singular` and `plural` and returns the appropriate string based on `number` and the `context` in a unicode string.

ngettext_lazy(*singular*, *plural*, *number*)

ungettext_lazy(*singular*, *plural*, *number*)

npgettext_lazy(*singular*, *plural*, *number*)
Same as the non-lazy versions above, but using lazy execution.

See *lazy translations documentation*.

string_concat(**strings*)
Lazy variant of string concatenation, needed for translations that are constructed from multiple parts.

activate(*language*)
Fetches the translation object for a given language and installs it as the current translation object for the current thread.

deactivate()
De-installs the currently active translation object so that further _ calls will resolve against the default translation object, again.

deactivate_all()
Makes the active translation object a NullTranslations() instance. This is useful when we want delayed translations to appear as the original string for some reason.

override(*language*, *deactivate=False*)
New in version 1.4: *Please see the release notes* A Python context manager that uses `django.utils.translation.activate()` to fetch the translation object for a given language, installing it as the translation object for the current thread and reinstall the previous active language on exit. Optionally it can simply deinstall the temporary translation on exit with `django.utils.translation.deactivate()` if the deactivate argument is True. If you pass None as the language argument, a NullTranslations() instance is installed while the context is active.

get_language()
Returns the currently selected language code.

get_language_bidi()
Returns selected language's BiDi layout:

- `False` = left-to-right layout
- `True` = right-to-left layout

get_language_from_request(*request*, *check_path=False*)
Changed in version 1.4: *Please see the release notes* Analyzes the request to find what language the user wants the system to show. Only languages listed in settings.LANGUAGES are taken into account. If the user requests a sublanguage where we have a main language, we send out the main language.

If `check_path` is `True`, the function first checks the requested URL for whether its path begins with a language code listed in the `LANGUAGES` setting.

to_locale(*language*)
Turns a language name (en-us) into a locale name (en_US).

templatize(*src*)
Turns a Django template into something that is understood by xgettext. It does so by translating the Django translation tags into standard gettext function invocations.

6.19.11 django.utils.timezone

New in version 1.4: *Please see the release notes*

utc
`tzinfo` instance that represents UTC.

get_default_timezone()
Returns a `tzinfo` instance that represents the *default time zone*.

get_default_timezone_name()
Returns the name of the *default time zone*.

get_current_timezone()
Returns a `tzinfo` instance that represents the *current time zone*.

get_current_timezone_name()
Returns the name of the *current time zone*.

activate(*timezone*)
Sets the *current time zone*. The `timezone` argument must be an instance of a `tzinfo` subclass or, if pytz is available, a time zone name.

deactivate()
Unsets the *current time zone*.

override(*timezone*)
This is a Python context manager that sets the *current time zone* on entry with `activate()`, and restores the

previously active time zone on exit. If the `timezone` argument is `None`, the *current time zone* is unset on entry with `deactivate()` instead.

New in version 1.5: *Please see the release notes*

localtime(*value*, *timezone=None*)
Converts an aware `datetime` to a different time zone, by default the *current time zone*.

This function doesn't work on naive datetimes; use `make_aware()` instead.

now()
Returns an aware or naive `datetime` that represents the current point in time when `USE_TZ` is `True` or `False` respectively.

is_aware(*value*)
Returns `True` if `value` is aware, `False` if it is naive. This function assumes that `value` is a `datetime`.

is_naive(*value*)
Returns `True` if `value` is naive, `False` if it is aware. This function assumes that `value` is a `datetime`.

make_aware(*value*, *timezone*)
Returns an aware `datetime` that represents the same point in time as `value` in `timezone`, `value` being a naive `datetime`.

This function can raise an exception if `value` doesn't exist or is ambiguous because of DST transitions.

make_naive(*value*, *timezone*)
Returns an naive `datetime` that represents in `timezone` the same point in time as `value`, `value` being an aware `datetime`

6.19.12 django.utils.tzinfo

class FixedOffset
Fixed offset in minutes east from UTC.

class LocalTimezone
Proxy timezone information from time module.

6.20 Validators

6.20.1 Writing validators

A validator is a callable that takes a value and raises a `ValidationError` if it doesn't meet some criteria. Validators can be useful for re-using validation logic between different types of fields.

For example, here's a validator that only allows even numbers:

```
from django.core.exceptions import ValidationError

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError(u'%s is not an even number' % value)
```

You can add this to a model field via the field's `validators` argument:

```
from django.db import models

class MyModel(models.Model):
    even_field = models.IntegerField(validators=[validate_even])
```

Because values are converted to Python before validators are run, you can even use the same validator with forms:

```
from django import forms

class MyForm(forms.Form):
    even_field = forms.IntegerField(validators=[validate_even])
```

6.20.2 How validators are run

See the *form validation* for more information on how validators are run in forms, and *Validating objects* for how they're run in models. Note that validators will not be run automatically when you save a model, but if you are using a `ModelForm`, it will run your validators on any fields that are included in your form. See the *ModelForm documentation* for information on how model validation interacts with forms.

6.20.3 Built-in validators

The `django.core.validators` module contains a collection of callable validators for use with model and form fields. They're used internally but are available for use with your own fields, too. They can be used in addition to, or in lieu of custom `field.clean()` methods.

RegexValidator

class RegexValidator([*regex=None*, *message=None*, *code=None*])

Parameters

- **regex** – If not `None`, overrides `regex`. Can be a regular expression string or a pre-compiled regular expression.
- **message** – If not `None`, overrides `message`.
- **code** – If not `None`, overrides `code`.

regex
The regular expression pattern to search for the provided `value`, or a pre-compiled regular expression. Raises a `ValidationError` with `message` and `code` if no match is found. By default, matches any string (including an empty string).

message
The error message used by `ValidationError` if validation fails. Defaults to `"Enter a valid value"`.

code
The error code used by `ValidationError` if validation fails. Defaults to `"invalid"`.

URLValidator

class URLValidator
A `RegexValidator` that ensures a value looks like a URL, and raises an error code of `'invalid'` if it doesn't.

validate_email

validate_email
A `RegexValidator` instance that ensures a value looks like an email address.

validate_slug

validate_slug
A `RegexValidator` instance that ensures a value consists of only letters, numbers, underscores or hyphens.

validate_ipv4_address

validate_ipv4_address
A `RegexValidator` instance that ensures a value looks like an IPv4 address.

validate_ipv6_address

New in version 1.4: *Please see the release notes*

validate_ipv6_address
Uses `django.utils.ipv6` to check the validity of an IPv6 address.

validate_ipv46_address

New in version 1.4: *Please see the release notes*

validate_ipv46_address
Uses both `validate_ipv4_address` and `validate_ipv6_address` to ensure a value is either a valid IPv4 or IPv6 address.

validate_comma_separated_integer_list

validate_comma_separated_integer_list
A `RegexValidator` instance that ensures a value is a comma-separated list of integers.

MaxValueValidator

class MaxValueValidator(*max_value*)
Raises a `ValidationError` with a code of `'max_value'` if `value` is greater than `max_value`.

MinValueValidator

class MinValueValidator(*min_value*)
Raises a `ValidationError` with a code of `'min_value'` if `value` is less than `min_value`.

MaxLengthValidator

class MaxLengthValidator(*max_length*)
Raises a `ValidationError` with a code of `'max_length'` if the length of `value` is greater than `max_length`.

MinLengthValidator

class **MinLengthValidator**(*min_length*)

Raises a `ValidationError` with a code of `'min_length'` if the length of `value` is less than `min_length`.

CHAPTER

SEVEN

META-DOCUMENTATION AND MISCELLANY

Documentation that we can't find a more organized place for. Like that drawer in your kitchen with the scissors, batteries, duct tape, and other junk.

7.1 API stability

The release of Django 1.0 comes with a promise of API stability and forwards-compatibility. In a nutshell, this means that code you develop against Django 1.0 will continue to work against 1.1 unchanged, and you should need to make only minor changes for any 1.X release.

7.1.1 What "stable" means

In this context, stable means:

- All the public APIs – everything documented in the linked documents below, and all methods that don't begin with an underscore – will not be moved or renamed without providing backwards-compatible aliases.
- If new features are added to these APIs – which is quite possible – they will not break or change the meaning of existing methods. In other words, "stable" does not (necessarily) mean "complete."
- If, for some reason, an API declared stable must be removed or replaced, it will be declared deprecated but will remain in the API for at least two minor version releases. Warnings will be issued when the deprecated method is called.

  See *Official releases* for more details on how Django's version numbering scheme works, and how features will be deprecated.
- We'll only break backwards compatibility of these APIs if a bug or security hole makes it completely unavoidable.

7.1.2 Stable APIs

In general, everything covered in the documentation – with the exception of anything in the *internals area* is considered stable as of 1.0. This includes these APIs:

- *Authorization*
- *Caching*.

- *Model definition, managers, querying and transactions*
- *Sending email*.
- *File handling and storage*
- *Forms*
- *HTTP request/response handling*, including file uploads, middleware, sessions, URL resolution, view, and shortcut APIs.
- *Generic views*.
- *Internationalization*.
- *Pagination*
- *Serialization*
- *Signals*
- *Templates*, including the language, Python-level *template APIs*, and *custom template tags and libraries*. We may add new template tags in the future and the names may inadvertently clash with external template tags. Before adding any such tags, we'll ensure that Django raises an error if it tries to load tags with duplicate names.
- *Testing*
- *django-admin utility*.
- *Built-in middleware*
- *Request/response objects*.
- *Settings*. Note, though that while the *list of built-in settings* can be considered complete we may – and probably will – add new settings in future versions. This is one of those places where "'stable' does not mean 'complete.'"
- *Built-in signals*. Like settings, we'll probably add new signals in the future, but the existing ones won't break.
- *Unicode handling*.
- Everything covered by the *HOWTO guides*.

`django.utils`

Most of the modules in `django.utils` are designed for internal use. Only the following parts of *django.utils* can be considered stable:

- `django.utils.cache`
- `django.utils.datastructures.SortedDict` – only this single class; the rest of the module is for internal use.
- `django.utils.encoding`
- `django.utils.feedgenerator`
- `django.utils.http`
- `django.utils.safestring`
- `django.utils.translation`
- `django.utils.tzinfo`

7.1.3 Exceptions

There are a few exceptions to this stability and backwards-compatibility promise.

Security fixes

If we become aware of a security problem – hopefully by someone following our *security reporting policy* – we'll do everything necessary to fix it. This might mean breaking backwards compatibility; security trumps the compatibility guarantee.

Contributed applications (`django.contrib`)

While we'll make every effort to keep these APIs stable – and have no plans to break any contrib apps – this is an area that will have more flux between releases. As the Web evolves, Django must evolve with it.

However, any changes to contrib apps will come with an important guarantee: we'll make sure it's always possible to use an older version of a contrib app if we need to make changes. Thus, if Django 1.5 ships with a backwards-incompatible `django.contrib.flatpages`, we'll make sure you can still use the Django 1.4 version alongside Django 1.5. This will continue to allow for easy upgrades.

Historically, apps in `django.contrib` have been more stable than the core, so in practice we probably won't have to ever make this exception. However, it's worth noting if you're building apps that depend on `django.contrib`.

APIs marked as internal

Certain APIs are explicitly marked as "internal" in a couple of ways:

- Some documentation refers to internals and mentions them as such. If the documentation says that something is internal, we reserve the right to change it.
- Functions, methods, and other objects prefixed by a leading underscore (_). This is the standard Python way of indicating that something is private; if any method starts with a single _, it's an internal API.

Local flavors

Changed in version 1.3: *Please see the release notes* `django.contrib.localflavor` contains assorted pieces of code that are useful for particular countries or cultures. This data is local in nature, and is subject to change on timelines that will almost never correlate with Django's own release schedules. For example, a common change is to split a province into two new provinces, or to rename an existing province.

These changes present two competing compatibility issues. Moving forward, displaying the names of deprecated, renamed and dissolved provinces in a selection widget is bad from a user interface perspective. However, maintaining full backwards compatibility requires that we support historical values that may be stored in a database – including values that may no longer be valid.

Therefore, Django has the following policy with respect to changes in local flavor:

- At the time of a Django release, the data and algorithms contained in `django.contrib.localflavor` will, to the best of our ability, reflect the officially gazetted policies of the appropriate local government authority. If a province has been added, altered, or removed, that change will be reflected in Django's localflavor.
- These changes will *not* be backported to the previous stable release. Upgrading a minor version of Django should not require any data migration or audits for UI changes; therefore, if you want to get the latest province list, you will either need to upgrade your Django install, or backport the province list you need.
- For one release, the affected localflavor module will raise a `RuntimeWarning` when it is imported.

- The change will be announced in the release notes as a backwards incompatible change requiring attention. The change will also be annotated in the documentation for the localflavor module.
- Where necessary and feasible, a migration script will be provided to aid the migration process.

For example, Django 1.2 contains an Indonesian localflavor. It has a province list that includes "Nanggroe Aceh Darussalam (NAD)" as a province. The Indonesian government has changed the official name of the province to "Aceh (ACE)". As a result, Django 1.3 does *not* contain "Nanggroe Aceh Darussalam (NAD)" in the province list, but *does* contain "Aceh (ACE)".

7.2 Design philosophies

This document explains some of the fundamental philosophies Django's developers have used in creating the framework. Its goal is to explain the past and guide the future.

7.2.1 Overall

Loose coupling

A fundamental goal of Django's stack is loose coupling and tight cohesion. The various layers of the framework shouldn't "know" about each other unless absolutely necessary.

For example, the template system knows nothing about Web requests, the database layer knows nothing about data display and the view system doesn't care which template system a programmer uses.

Although Django comes with a full stack for convenience, the pieces of the stack are independent of another wherever possible.

Less code

Django apps should use as little code as possible; they should lack boilerplate. Django should take full advantage of Python's dynamic capabilities, such as introspection.

Quick development

The point of a Web framework in the 21st century is to make the tedious aspects of Web development fast. Django should allow for incredibly quick Web development.

Don't repeat yourself (DRY)

Every distinct concept and/or piece of data should live in one, and only one, place. Redundancy is bad. Normalization is good.

The framework, within reason, should deduce as much as possible from as little as possible.

See Also:

The discussion of DRY on the Portland Pattern Repository

Explicit is better than implicit

This is a core Python principle listed in PEP 20, and it means Django shouldn't do too much "magic." Magic shouldn't happen unless there's a really good reason for it. Magic is worth using only if it creates a huge convenience unattainable in other ways, and it isn't implemented in a way that confuses developers who are trying to learn how to use the feature.

Consistency

The framework should be consistent at all levels. Consistency applies to everything from low-level (the Python coding style used) to high-level (the "experience" of using Django).

7.2.2 Models

Explicit is better than implicit

Fields shouldn't assume certain behaviors based solely on the name of the field. This requires too much knowledge of the system and is prone to errors. Instead, behaviors should be based on keyword arguments and, in some cases, on the type of the field.

Include all relevant domain logic

Models should encapsulate every aspect of an "object," following Martin Fowler's Active Record design pattern.

This is why both the data represented by a model and information about it (its human-readable name, options like default ordering, etc.) are defined in the model class; all the information needed to understand a given model should be stored *in* the model.

7.2.3 Database API

The core goals of the database API are:

SQL efficiency

It should execute SQL statements as few times as possible, and it should optimize statements internally.

This is why developers need to call `save()` explicitly, rather than the framework saving things behind the scenes silently.

This is also why the `select_related()` `QuerySet` method exists. It's an optional performance booster for the common case of selecting "every related object."

Terse, powerful syntax

The database API should allow rich, expressive statements in as little syntax as possible. It should not rely on importing other modules or helper objects.

Joins should be performed automatically, behind the scenes, when necessary.

Every object should be able to access every related object, systemwide. This access should work both ways.

Option to drop into raw SQL easily, when needed

The database API should realize it's a shortcut but not necessarily an end-all-be-all. The framework should make it easy to write custom SQL – entire statements, or just custom `WHERE` clauses as custom parameters to API calls.

7.2.4 URL design

Loose coupling

URLs in a Django app should not be coupled to the underlying Python code. Tying URLs to Python function names is a Bad And Ugly Thing.

Along these lines, the Django URL system should allow URLs for the same app to be different in different contexts. For example, one site may put stories at `/stories/`, while another may use `/news/`.

Infinite flexibility

URLs should be as flexible as possible. Any conceivable URL design should be allowed.

Encourage best practices

The framework should make it just as easy (or even easier) for a developer to design pretty URLs than ugly ones.

File extensions in Web-page URLs should be avoided.

Vignette-style commas in URLs deserve severe punishment.

Definitive URLs

Technically, `foo.com/bar` and `foo.com/bar/` are two different URLs, and search-engine robots (and some Web traffic-analyzing tools) would treat them as separate pages. Django should make an effort to "normalize" URLs so that search-engine robots don't get confused.

This is the reasoning behind the `APPEND_SLASH` setting.

7.2.5 Template system

Separate logic from presentation

We see a template system as a tool that controls presentation and presentation-related logic – and that's it. The template system shouldn't support functionality that goes beyond this basic goal.

If we wanted to put everything in templates, we'd be using PHP. Been there, done that, wised up.

Discourage redundancy

The majority of dynamic Web sites use some sort of common sitewide design – a common header, footer, navigation bar, etc. The Django template system should make it easy to store those elements in a single place, eliminating duplicate code.

This is the philosophy behind *template inheritance*.

Be decoupled from HTML

The template system shouldn't be designed so that it only outputs HTML. It should be equally good at generating other text-based formats, or just plain text.

XML should not be used for template languages

Using an XML engine to parse templates introduces a whole new world of human error in editing templates – and incurs an unacceptable level of overhead in template processing.

Assume designer competence

The template system shouldn't be designed so that templates necessarily are displayed nicely in WYSIWYG editors such as Dreamweaver. That is too severe of a limitation and wouldn't allow the syntax to be as nice as it is. Django expects template authors are comfortable editing HTML directly.

Treat whitespace obviously

The template system shouldn't do magic things with whitespace. If a template includes whitespace, the system should treat the whitespace as it treats text – just display it. Any whitespace that's not in a template tag should be displayed.

Don't invent a programming language

The template system intentionally doesn't allow the following:

- Assignment to variables
- Advanced logic

The goal is not to invent a programming language. The goal is to offer just enough programming-esque functionality, such as branching and looping, that is essential for making presentation-related decisions.

The Django template system recognizes that templates are most often written by *designers*, not *programmers*, and therefore should not assume Python knowledge.

Safety and security

The template system, out of the box, should forbid the inclusion of malicious code – such as commands that delete database records.

This is another reason the template system doesn't allow arbitrary Python code.

Extensibility

The template system should recognize that advanced template authors may want to extend its technology.

This is the philosophy behind custom template tags and filters.

7.2.6 Views

Simplicity

Writing a view should be as simple as writing a Python function. Developers shouldn't have to instantiate a class when a function will do.

Use request objects

Views should have access to a request object – an object that stores metadata about the current request. The object should be passed directly to a view function, rather than the view function having to access the request data from a global variable. This makes it light, clean and easy to test views by passing in "fake" request objects.

Loose coupling

A view shouldn't care about which template system the developer uses – or even whether a template system is used at all.

Differentiate between GET and POST

GET and POST are distinct; developers should explicitly use one or the other. The framework should make it easy to distinguish between GET and POST data.

7.3 Third-party distributions of Django

Many third-party distributors are now providing versions of Django integrated with their package-management systems. These can make installation and upgrading much easier for users of Django since the integration includes the ability to automatically install dependencies (like database adapters) that Django requires.

Typically, these packages are based on the latest stable release of Django, so if you want to use the development version of Django you'll need to follow the instructions for *installing the development version* from our Subversion repository.

If you're using Linux or a Unix installation, such as OpenSolaris, check with your distributor to see if they already package Django. If you're using a Linux distro and don't know how to find out if a package is available, then now is a good time to learn. The Django Wiki contains a list of Third Party Distributions to help you out.

7.3.1 For distributors

If you'd like to package Django for distribution, we'd be happy to help out! Please join the django-developers mailing list and introduce yourself.

We also encourage all distributors to subscribe to the django-announce mailing list, which is a (very) low-traffic list for announcing new releases of Django and important bugfixes.

CHAPTER

EIGHT

GLOSSARY

field An attribute on a *model*; a given field usually maps directly to a single database column.

See *Models*.

generic view A higher-order *view* function that provides an abstract/generic implementation of a common idiom or pattern found in view development.

See *Class-based views*.

model Models store your application's data.

See *Models*.

MTV "Model-template-view"; a software pattern, similar in style to MVC, but a better description of the way Django does things.

See *the FAQ entry*.

MVC Model-view-controller; a software pattern. Django *follows MVC to some extent*.

project A Python package – i.e. a directory of code – that contains all the settings for an instance of Django. This would include database configuration, Django-specific options and application-specific settings.

property Also known as "managed attributes", and a feature of Python since version 2.2. This is a neat way to implement attributes whose usage resembles attribute access, but whose implementation uses method calls.

See `property()`.

queryset An object representing some set of rows to be fetched from the database.

See *Making queries*.

slug A short label for something, containing only letters, numbers, underscores or hyphens. They're generally used in URLs. For example, in a typical blog entry URL:

```
https://www.djangoproject.com/weblog/2008/apr/12/spring/
```

the last bit (`spring`) is the slug.

template A chunk of text that acts as formatting for representing data. A template helps to abstract the presentation of data from the data itself.

See *The Django template language*.

view A function responsible for rending a page.

CHAPTER

NINE

RELEASE NOTES

Release notes for the official Django releases. Each release note will tell you what's new in each version, and will also describe any backwards-incompatible changes made in that version.

For those upgrading to a new version of Django, you will need to check all the backwards-incompatible changes and deprecated features for each 'final' release from the one after your current Django version, up to and including the new version.

9.1 Final releases

9.1.1 1.5 release

Django 1.5 release notes - UNDER DEVELOPMENT

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from Django 1.4 or older versions. We've also dropped some features, which are detailed in *our deprecation plan*, and we've begun the deprecation process for some features.

Python compatibility

Django 1.5 has dropped support for Python 2.5. Python 2.6.5 is now the minimum required Python version. Django is tested and supported on Python 2.6 and 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.6 or newer as their default version. If you're still using Python 2.5, however, you'll need to stick to Django 1.4 until you can upgrade your Python version. Per *our support policy*, Django 1.4 will continue to receive security support until the release of Django 1.6.

Django 1.5 does not run on a Jython final release, because Jython's latest release doesn't currently support Python 2.6. However, Jython currently does offer an alpha release featuring 2.7 support.

What's new in Django 1.5

Support for saving a subset of model's fields The method `Model.save()` has a new keyword argument `update_fields`. By using this argument it is possible to save only a select list of model's fields. This can be useful for performance reasons or when trying to avoid overwriting concurrent changes.

See the `Model.save()` documentation for more details.

Caching of related model instances When traversing relations, the ORM will avoid re-fetching objects that were previously loaded. For example, with the tutorial's models:

```
>>> first_poll = Poll.objects.all()[0]
>>> first_choice = first_poll.choice_set.all()[0]
>>> first_choice.poll is first_poll
True
```

In Django 1.5, the third line no longer triggers a new SQL query to fetch `first_choice.poll`; it was set by the second line.

For one-to-one relationships, both sides can be cached. For many-to-one relationships, only the single side of the relationship can be cached. This is particularly helpful in combination with `prefetch_related`.

`{% verbatim %}` template tag To make it easier to deal with javascript templates which collide with Django's syntax, you can now use the `verbatim` block tag to avoid parsing the tag's content.

Retrieval of `ContentType` instances associated with proxy models The methods `ContentTypeManager.get_for_model()` and `ContentTypeManager.get_for_models()` have a new keyword argument – respectively `for_concrete_model` and `for_concrete_models`. By passing `False` using this argument it is now possible to retreive the `ContentType` associated with proxy models.

Minor features Django 1.5 also includes several smaller improvements worth noting:

- The template engine now interprets `True`, `False` and `None` as the corresponding Python objects.
- `django.utils.timezone` provides a helper for converting aware datetimes between time zones. See `localtime()`.
- The generic views support OPTIONS requests.
- Management commands do not raise `SystemExit` any more when called by code from *call_command*. Any exception raised by the command (mostly *CommandError*) is propagated.
- The dumpdata management command outputs one row at a time, preventing out-of-memory errors when dumping large datasets.
- In the localflavor for Canada, "pq" was added to the acceptable codes for Quebec. It's an old abbreviation.
- The *receiver* decorator is now able to connect to more than one signal by supplying a list of signals.
- `QuerySet.bulk_create()` has now a batch_size argument. By default the batch_size is unlimited except for SQLite where single batch is limited so that 999 parameters per query isn't exceeded.

Backwards incompatible changes in 1.5

> **Warning:** In addition to the changes outlined in this section, be sure to review the *deprecation plan* for any features that have been removed. If you haven't updated your code within the deprecation timeline for a given feature, its removal may appear as a backwards incompatible change.

Context in year archive class-based views For consistency with the other date-based generic views, `YearArchiveView` now passes `year` in the context as a `datetime.date` rather than a string. If you are using `{{ year }}` in your templates, you must replace it with `{{ year|date:"Y" }}`.

`next_year` and `previous_year` were also added in the context. They are calculated according to `allow_empty` and `allow_future`.

OPTIONS, PUT and DELETE requests in the test client Unlike GET and POST, these HTTP methods aren't implemented by web browsers. Rather, they're used in APIs, which transfer data in various formats such as JSON or XML. Since such requests may contain arbitrary data, Django doesn't attempt to decode their body.

However, the test client used to build a query string for OPTIONS and DELETE requests like for GET, and a request body for PUT requests like for POST. This encoding was arbitrary and inconsistent with Django's behavior when it receives the requests, so it was removed in Django 1.5.

If you were using the `data` parameter in an OPTIONS or a DELETE request, you must convert it to a query string and append it to the `path` parameter.

If you were using the `data` parameter in a PUT request without a `content_type`, you must encode your data before passing it to the test client and set the `content_type` argument.

String types of hasher method parameters If you have written a *custom password hasher*, your `encode()`, `verify()` or `safe_summary()` methods should accept Unicode parameters (`password`, `salt` or `encoded`). If any of the hashing methods need byte strings, you can use the `smart_str()` utility to encode the strings.

Validation of previous_page_number and next_page_number When using *object pagination*, the `previous_page_number()` and `next_page_number()` methods of the `Page` object did not check if the returned number was inside the existing page range. It does check it now and raises an `InvalidPage` exception when the number is either too low or too high.

Behavior of autocommit database option on PostgreSQL changed PostgreSQL's autocommit option didn't work as advertised previously. It did work for single transaction block, but after the first block was left the autocommit behavior was never restored. This bug is now fixed in 1.5. While this is only a bug fix, it is worth checking your applications behavior if you are using PostgreSQL together with the autocommit option.

Session not saved on 500 responses Django's session middleware will skip saving the session data if the response's status code is 500.

Changes in tests execution Some changes have been introduced in the execution of tests that might be backward-incompatible for some testing setups:

Database flushing in `django.test.TransactionTestCase` Previously, the test database was truncated *before* each test run in a `TransactionTestCase`.

In order to be able to run unit tests in any order and to make sure they are always isolated from each other, `TransactionTestCase` will now reset the database *after* each test run instead.

No more implict DB sequences reset `TransactionTestCase` tests used to reset primary key sequences automatically together with the database flushing actions described above.

This has been changed so no sequences are implicitly reset. This can cause `TransactionTestCase` tests that depend on hard-coded primary key values to break.

The new `reset_sequences` attribute can be used to force the old behavior for `TransactionTestCase` that might need it.

Ordering of tests In order to make sure all `TestCase` code starts with a clean database, tests are now executed in the following order:

- First, all unittests (including `unittest.TestCase`, `SimpleTestCase`, `TestCase` and `TransactionTestCase`) are run with no particular ordering guaranteed nor enforced among them.
- Then any other tests (e.g. doctests) that may alter the database without restoring it to its original state are run.

This should not cause any problems unless you have existing doctests which assume a `TransactionTestCase` executed earlier left some database state behind or unit tests that rely on some form of state being preserved after the execution of other tests. Such tests are already very fragile, and must now be changed to be able to run independently.

Miscellaneous

- GeoDjango dropped support for GDAL < 1.5

Features deprecated in 1.5

`django.utils.simplejson` Since Django 1.5 drops support for Python 2.5, we can now rely on the `json` module being in Python's standard library – so we've removed our own copy of `simplejson`. You can safely change any use of `django.utils.simplejson` to `json`.

`itercompat.product` The `product()` function has been deprecated. Use the built-in `itertools.product()` instead.

9.1.2 1.4 release

Django 1.4 release notes

March 23, 2012

Welcome to Django 1.4!

These release notes cover the new features, as well as some backwards incompatible changes you'll want to be aware of when upgrading from Django 1.3 or older versions. We've also dropped some features, which are detailed in *our deprecation plan*, and we've begun the deprecation process for some features.

Overview

The biggest new feature in Django 1.4 is support for time zones when handling date/times. When enabled, this Django will store date/times in UTC, use timezone-aware objects internally, and translate them to users' local timezones for display.

If you're upgrading an existing project to Django 1.4, switching to the time- zone aware mode may take some care: the new mode disallows some rather sloppy behavior that used to be accepted. We encourage anyone who's upgrading to check out the *timezone migration guide* and the *timezone FAQ* for useful pointers.

Other notable new features in Django 1.4 include:

- A number of ORM improvements, including SELECT FOR UPDATE support, the ability to bulk insert large datasets for improved performance, and QuerySet.prefetch_related, a method to batch-load related objects in areas where `select_related()` doesn't work.
- Some nice security additions, including improved password hashing (featuring PBKDF2 and bcrypt support), new tools for cryptographic signing, several CSRF improvements, and simple clickjacking protection.

- An updated default project layout and manage.py that removes the "magic" from prior versions. And for those who don't like the new layout, you can use custom project and app templates instead!
- Support for in-browser testing frameworks (like Selenium).
- ... and a whole lot more; see below!

Wherever possible we try to introduce new features in a backwards-compatible manner per *our API stability policy* policy. However, as with previous releases, Django 1.4 ships with some minor backwards incompatible changes; people upgrading from previous versions of Django should read that list carefully.

Python compatibility

Django 1.4 has dropped support for Python 2.4. Python 2.5 is now the minimum required Python version. Django is tested and supported on Python 2.5, 2.6 and 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.5 or newer as their default version. If you're still using Python 2.4, however, you'll need to stick to Django 1.3 until you can upgrade. Per *our support policy*, Django 1.3 will continue to receive security support until the release of Django 1.5.

Django does not support Python 3.x at this time. At some point before the release of Django 1.4, we plan to publish a document outlining our full timeline for deprecating Python 2.x and moving to Python 3.x.

What's new in Django 1.4

Support for time zones In previous versions, Django used "naive" date/times (that is, date/times without an associated time zone), leaving it up to each developer to interpret what a given date/time "really means". This can cause all sorts of subtle timezone-related bugs.

In Django 1.4, you can now switch Django into a more correct, time-zone aware mode. In this mode, Django stores date and time information in UTC in the database, uses time-zone-aware datetime objects internally and translates them to the end user's time zone in templates and forms. Reasons for using this feature include:

- Customizing date and time display for users around the world.
- Storing datetimes in UTC for database portability and interoperability. (This argument doesn't apply to PostgreSQL, because it already stores timestamps with time zone information in Django 1.3.)
- Avoiding data corruption problems around DST transitions.

Time zone support is enabled by default in new projects created with `startproject`. If you want to use this feature in an existing project, read the *migration guide*. If you encounter problems, there's a helpful *FAQ*.

Support for in-browser testing frameworks Django 1.4 supports integration with in-browser testing frameworks like Selenium. The new `django.test.LiveServerTestCase` base class lets you test the interactions between your site's front and back ends more comprehensively. See the `documentation` for more details and concrete examples.

Updated default project layout and `manage.py` Django 1.4 ships with an updated default project layout and `manage.py` file for the `startproject` management command. These fix some issues with the previous `manage.py` handling of Python import paths that caused double imports, trouble moving from development to deployment, and other difficult-to-debug path issues.

The previous `manage.py` called functions that are now deprecated, and thus projects upgrading to Django 1.4 should update their `manage.py`. (The old-style `manage.py` will continue to work as before until Django 1.6. In 1.5 it will raise `DeprecationWarning`).

The new recommended `manage.py` file should look like this:

```
#!/usr/bin/env python
import os, sys

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "{{ project_name }}.settings")

    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

`{{ project_name }}` should be replaced with the Python package name of the actual project.

If settings, URLconfs and apps within the project are imported or referenced using the project name prefix (e.g. `myproject.settings`, `ROOT_URLCONF = "myproject.urls"`, etc), the new `manage.py` will need to be moved one directory up, so it is outside the project package rather than adjacent to `settings.py` and `urls.py`.

For instance, with the following layout:

```
manage.py
mysite/
    __init__.py
    settings.py
    urls.py
    myapp/
        __init__.py
        models.py
```

You could import `mysite.settings`, `mysite.urls`, and `mysite.myapp`, but not `settings`, `urls`, or `myapp` as top-level modules.

Anything imported as a top-level module can be placed adjacent to the new `manage.py`. For instance, to decouple "myapp" from the project module and import it as just `myapp`, place it outside the `mysite/` directory:

```
manage.py
myapp/
    __init__.py
    models.py
mysite/
    __init__.py
    settings.py
    urls.py
```

If the same code is imported inconsistently (some places with the project prefix, some places without it), the imports will need to be cleaned up when switching to the new `manage.py`.

Custom project and app templates The `startapp` and `startproject` management commands now have a `--template` option for specifying a path or URL to a custom app or project template.

For example, Django will use the `/path/to/my_project_template` directory when you run the following command:

```
django-admin.py startproject --template=/path/to/my_project_template myproject
```

You can also now provide a destination directory as the second argument to both `startapp` and `startproject`:

```
django-admin.py startapp myapp /path/to/new/app
django-admin.py startproject myproject /path/to/new/project
```

For more information, see the `startapp` and `startproject` documentation.

Improved WSGI support The `startproject` management command now adds a `wsgi.py` module to the initial project layout, containing a simple WSGI application that can be used for *deploying with WSGI app servers*.

The `built-in development server` now supports using an externally-defined WSGI callable, which makes it possible to run runserver with the same WSGI configuration that is used for deployment. The new `WSGI_APPLICATION` setting lets you configure which WSGI callable `runserver` uses.

(The `runfcgi` management command also internally wraps the WSGI callable configured via `WSGI_APPLICATION`.)

`SELECT FOR UPDATE` support Django 1.4 includes a `QuerySet.select_for_update()` method, which generates a `SELECT ... FOR UPDATE` SQL query. This will lock rows until the end of the transaction, meaning other transactions cannot modify or delete rows matched by a `FOR UPDATE` query.

For more details, see the documentation for `select_for_update()`.

`Model.objects.bulk_create` in the ORM This method lets you create multiple objects more efficiently. It can result in significant performance increases if you have many objects.

Django makes use of this internally, meaning some operations (such as database setup for test suites) have seen a performance benefit as a result.

See the `bulk_create()` docs for more information.

`QuerySet.prefetch_related` Similar to `select_related()` but with a different strategy and broader scope, `prefetch_related()` has been added to `QuerySet`. This method returns a new `QuerySet` that will prefetch each of the specified related lookups in a single batch as soon as the query begins to be evaluated. Unlike `select_related`, it does the joins in Python, not in the database, and supports many-to-many relationships, `GenericForeignKey` and more. This allows you to fix a very common performance problem in which your code ends up doing O(n) database queries (or worse) if objects on your primary `QuerySet` each have many related objects that you also need to fetch.

Improved password hashing Django's auth system (`django.contrib.auth`) stores passwords using a one-way algorithm. Django 1.3 uses the SHA1 algorithm, but increasing processor speeds and theoretical attacks have revealed that SHA1 isn't as secure as we'd like. Thus, Django 1.4 introduces a new password storage system: by default Django now uses the PBKDF2 algorithm (as recommended by NIST). You can also easily choose a different algorithm (including the popular bcrypt algorithm). For more details, see *How Django stores passwords*.

HTML5 doctype We've switched the admin and other bundled templates to use the HTML5 doctype. While Django will be careful to maintain compatibility with older browsers, this change means that you can use any HTML5 features you need in admin pages without having to lose HTML validity or override the provided templates to change the doctype.

List filters in admin interface Prior to Django 1.4, the `admin` app let you specify change list filters by specifying a field lookup, but it didn't allow you to create custom filters. This has been rectified with a simple API (previously used internally and known as "FilterSpec"). For more details, see the documentation for `list_filter`.

Multiple sort in admin interface The admin change list now supports sorting on multiple columns. It respects all elements of the `ordering` attribute, and sorting on multiple columns by clicking on headers is designed to mimic the behavior of desktop GUIs. We also added a `get_ordering()` method for specifying the ordering dynamically (i.e., depending on the request).

New `ModelAdmin` methods We added a `save_related()` method to `ModelAdmin` to ease customization of how related objects are saved in the admin.

Two other new `ModelAdmin` methods, `get_list_display()` and `get_list_display_links()` enable dynamic customization of fields and links displayed on the admin change list.

Admin inlines respect user permissions Admin inlines now only allow those actions for which the user has permission. For `ManyToMany` relationships with an auto-created intermediate model (which does not have its own permissions), the change permission for the related model determines if the user has the permission to add, change or delete relationships.

Tools for cryptographic signing Django 1.4 adds both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in Web applications.

See the *cryptographic signing* docs for more information.

Cookie-based session backend Django 1.4 introduces a cookie-based session backend that uses the tools for *cryptographic signing* to store the session data in the client's browser.

> **Warning:** Session data is signed and validated by the server, but it's not encrypted. This means a user can view any data stored in the session but cannot change it. Please read the documentation for further clarification before using this backend.

See the *cookie-based session backend* docs for more information.

New form wizard The previous `FormWizard` from `django.contrib.formtools` has been replaced with a new implementation based on the class-based views introduced in Django 1.3. It features a pluggable storage API and doesn't require the wizard to pass around hidden fields for every previous step.

Django 1.4 ships with a session-based storage backend and a cookie-based storage backend. The latter uses the tools for *cryptographic signing* also introduced in Django 1.4 to store the wizard's state in the user's cookies.

See the *form wizard* docs for more information.

`reverse_lazy` A lazily evaluated version of `django.core.urlresolvers.reverse()` was added to allow using URL reversals before the project's URLconf gets loaded.

Translating URL patterns Django can now look for a language prefix in the URLpattern when using the new `i18n_patterns()` helper function. It's also now possible to define translatable URL patterns using `ugettext_lazy()`. See *Internationalization: in URL patterns* for more information about the language prefix and how to internationalize URL patterns.

Contextual translation support for `{% trans %}` and `{% blocktrans %}` The *contextual translation* support introduced in Django 1.3 via the `pgettext` function has been extended to the `trans` and `blocktrans` template tags using the new `context` keyword.

Customizable `SingleObjectMixin` URLConf kwargs Two new attributes, `pk_url_kwarg` and `slug_url_kwarg`, have been added to `SingleObjectMixin` to enable the customization of URLconf keyword arguments used for single object generic views.

Assignment template tags A new *assignment_tag* helper function was added to `template.Library` to ease the creation of template tags that store data in a specified context variable.

`*args` and `**kwargs` support for template tag helper functions The *simple_tag*, *inclusion_tag* and newly introduced *assignment_tag* template helper functions may now accept any number of positional or keyword arguments. For example:

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then, in the template, any number of arguments may be passed to the template tag. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

No wrapping of exceptions in `TEMPLATE_DEBUG` mode In previous versions of Django, whenever the `TEMPLATE_DEBUG` setting was `True`, any exception raised during template rendering (even exceptions unrelated to template syntax) were wrapped in `TemplateSyntaxError` and re-raised. This was done in order to provide detailed template source location information in the debug 500 page.

In Django 1.4, exceptions are no longer wrapped. Instead, the original exception is annotated with the source information. This means that catching exceptions from template rendering is now consistent regardless of the value of `TEMPLATE_DEBUG`, and there's no need to catch and unwrap `TemplateSyntaxError` in order to catch other errors.

`truncatechars` template filter This new filter truncates a string to be no longer than the specified number of characters. Truncated strings end with a translatable ellipsis sequence ("..."). See the documentation for `truncatechars` for more details.

`static` template tag The `staticfiles` contrib app has a new `static` template tag to refer to files saved with the `STATICFILES_STORAGE` storage backend. It uses the storage backend's `url` method and therefore supports advanced features such as *serving files from a cloud service*.

`CachedStaticFilesStorage` storage backend The `staticfiles` contrib app now has a `CachedStaticFilesStorage` backend that caches the files it saves (when running the `collectstatic` management command) by appending the MD5 hash of the file's content to the filename. For example, the file `css/styles.css` would also be saved as `css/styles.55e7cbb9ba48.css`

See the `CachedStaticFilesStorage` docs for more information.

Simple clickjacking protection We've added a middleware to provide easy protection against clickjacking using the `X-Frame-Options` header. It's not enabled by default for backwards compatibility reasons, but you'll almost certainly want to *enable it* to help plug that security hole for browsers that support the header.

CSRF improvements We've made various improvements to our CSRF features, including the `ensure_csrf_cookie()` decorator, which can help with AJAX-heavy sites; protection for PUT and DELETE requests; and the `CSRF_COOKIE_SECURE` and `CSRF_COOKIE_PATH` settings, which can improve the security and usefulness of CSRF protection. See the *CSRF docs* for more information.

Error report filtering We added two function decorators, `sensitive_variables()` and `sensitive_post_parameters()`, to allow designating the local variables and POST parameters that may contain sensitive information and should be filtered out of error reports.

All POST parameters are now systematically filtered out of error reports for certain views (`login`, `password_reset_confirm`, `password_change` and `add_view` in `django.contrib.auth.views`, as well as `user_change_password` in the admin app) to prevent the leaking of sensitive information such as user passwords.

You can override or customize the default filtering by writing a *custom filter*. For more information see the docs on *Filtering error reports*.

Extended IPv6 support Django 1.4 can now better handle IPv6 addresses with the new `GenericIPAddressField` model field, `GenericIPAddressField` form field and the validators `validate_ipv46_address` and `validate_ipv6_address`.

HTML comparisons in tests The base classes in `django.test` now have some helpers to compare HTML without tripping over irrelevant differences in whitespace, argument quoting/ordering and closing of self-closing tags. You can either compare HTML directly with the new `assertHTMLEqual()` and `assertHTMLNotEqual()` assertions, or use the `html=True` flag with `assertContains()` and `assertNotContains()` to test whether the client's response contains a given HTML fragment. See the *assertions documentation* for more.

Two new date format strings Two new `date` formats were added for use in template filters, template tags and *Format localization*:

- `e` – the name of the timezone of the given datetime object
- `o` – the ISO 8601 year number

Please make sure to update your *custom format files* if they contain either `e` or `o` in a format string. For example a Spanish localization format previously only escaped the `d` format character:

```
DATE_FORMAT = r'j \de F \de Y'
```

But now it needs to also escape `e` and `o`:

```
DATE_FORMAT = r'j \d\e F \d\e Y'
```

For more information, see the `date` documentation.

Minor features Django 1.4 also includes several smaller improvements worth noting:

- A more usable stacktrace in the technical 500 page. Frames in the stack trace that reference Django's framework code are dimmed out, while frames in application code are slightly emphasized. This change makes it easier to scan a stacktrace for issues in application code.
- *Tablespace support* in PostgreSQL.
- Customizable names for `simple_tag()`.
- In the documentation, a helpful *security overview* page.

- The `django.contrib.auth.models.check_password()` function has been moved to the `django.contrib.auth.hashers` module. Importing it from the old location will still work, but you should update your imports.
- The `collectstatic` management command now has a `--clear` option to delete all files at the destination before copying or linking the static files.
- It's now possible to load fixtures containing forward references when using MySQL with the InnoDB database engine.
- A new 403 response handler has been added as `'django.views.defaults.permission_denied'`. You can set your own handler by setting the value of `django.conf.urls.handler403`. See the documentation about *the 403 (HTTP Forbidden) view* for more information.
- The `makemessages` command uses a new and more accurate lexer, JsLex, for extracting translatable strings from JavaScript files.
- The `trans` template tag now takes an optional `as` argument to be able to retrieve a translation string without displaying it but setting a template context variable instead.
- The `if` template tag now supports `{% elif %}` clauses.
- If your Django app is behind a proxy, you might find the new `SECURE_PROXY_SSL_HEADER` setting useful. It solves the problem of your proxy "eating" the fact that a request came in via HTTPS. But only use this setting if you know what you're doing.
- A new, plain-text, version of the HTTP 500 status code internal error page served when `DEBUG` is `True` is now sent to the client when Django detects that the request has originated in JavaScript code. (`is_ajax()` is used for this.)

  Like its HTML counterpart, it contains a collection of different pieces of information about the state of the application.

  This should make it easier to read when debugging interaction with client-side JavaScript.
- Added the `--no-location` option to the `makemessages` command.
- Changed the `locmem` cache backend to use `pickle.HIGHEST_PROTOCOL` for better compatibility with the other cache backends.
- Added support in the ORM for generating `SELECT` queries containing `DISTINCT ON`.

  The `distinct()` `QuerySet` method now accepts an optional list of model field names. If specified, then the `DISTINCT` statement is limited to these fields. This is only supported in PostgreSQL.

  For more details, see the documentation for `distinct()`.
- The admin login page will add a password reset link if you include a URL with the name *'admin_password_reset'* in your urls.py, so plugging in the built-in password reset mechanism and making it available is now much easier. For details, see *Adding a password-reset feature*.
- The MySQL database backend can now make use of the savepoint feature implemented by MySQL version 5.0.3 or newer with the InnoDB storage engine.
- It's now possible to pass initial values to the model forms that are part of both model formsets and inline model formsets as returned from factory functions `modelformset_factory` and `inlineformset_factory` respectively just like with regular formsets. However, initial values only apply to extra forms, i.e. those which are not bound to an existing model instance.
- The sitemaps framework can now handle HTTPS links using the new `Sitemap.protocol` class attribute.
- A new `django.test.SimpleTestCase` subclass of `unittest.TestCase` that's lighter than `django.test.TestCase` and company. It can be useful in tests that don't need to hit a database. See *Hierarchy of Django unit testing classes*.

Backwards incompatible changes in 1.4

SECRET_KEY setting is required Running Django with an empty or known `SECRET_KEY` disables many of Django's security protections and can lead to remote-code-execution vulnerabilities. No Django site should ever be run without a `SECRET_KEY`.

In Django 1.4, starting Django with an empty `SECRET_KEY` will raise a *DeprecationWarning*. In Django 1.5, it will raise an exception and Django will refuse to start. This is slightly accelerated from the usual deprecation path due to the severity of the consequences of running Django with no `SECRET_KEY`.

django.contrib.admin The included administration app `django.contrib.admin` has for a long time shipped with a default set of static files such as JavaScript, images and stylesheets. Django 1.3 added a new contrib app `django.contrib.staticfiles` to handle such files in a generic way and defined conventions for static files included in apps.

Starting in Django 1.4, the admin's static files also follow this convention, to make the files easier to deploy. In previous versions of Django, it was also common to define an `ADMIN_MEDIA_PREFIX` setting to point to the URL where the admin's static files live on a Web server. This setting has now been deprecated and replaced by the more general setting `STATIC_URL`. Django will now expect to find the admin static files under the URL `<STATIC_URL>/admin/`.

If you've previously used a URL path for `ADMIN_MEDIA_PREFIX` (e.g. `/media/`) simply make sure `STATIC_URL` and `STATIC_ROOT` are configured and your Web server serves those files correctly. The development server continues to serve the admin files just like before. Read the *static files howto* for more details.

If your `ADMIN_MEDIA_PREFIX` is set to an specific domain (e.g. `http://media.example.com/admin/`), make sure to also set your `STATIC_URL` setting to the correct URL – for example, `http://media.example.com/`.

> **Warning:** If you're implicitly relying on the path of the admin static files within Django's source code, you'll need to update that path. The files were moved from `django/contrib/admin/media/` to `django/contrib/admin/static/admin/`.

Supported browsers for the admin Django hasn't had a clear policy on which browsers are supported by the admin app. Our new policy formalizes existing practices: YUI's A-grade browsers should provide a fully-functional admin experience, with the notable exception of Internet Explorer 6, which is no longer supported.

Released over 10 years ago, IE6 imposes many limitations on modern Web development. The practical implications of this policy are that contributors are free to improve the admin without consideration for these limitations.

Obviously, this new policy **has no impact** on sites you develop using Django. It only applies to the Django admin. Feel free to develop apps compatible with any range of browsers.

Removed admin icons As part of an effort to improve the performance and usability of the admin's change-list sorting interface and `horizontal` and `vertical` "filter" widgets, some icon files were removed and grouped into two sprite files.

Specifically: `selector-add.gif`, `selector-addall.gif`, `selector-remove.gif`, `selector-removeall.gif`, `selector_stacked-add.gif` and `selector_stacked-remove.gif` were combined into `selector-icons.gif`; and `arrow-up.gif` and `arrow-down.gif` were combined into `sorting-icons.gif`.

If you used those icons to customize the admin, then you'll need to replace them with your own icons or get the files from a previous release.

CSS class names in admin forms To avoid conflicts with other common CSS class names (e.g. "button"), we added a prefix ("field-") to all CSS class names automatically generated from the form field names in the main admin forms, stacked inline forms and tabular inline cells. You'll need to take that prefix into account in your custom style sheets or JavaScript files if you previously used plain field names as selectors for custom styles or JavaScript transformations.

Compatibility with old signed data Django 1.3 changed the cryptographic signing mechanisms used in a number of places in Django. While Django 1.3 kept fallbacks that would accept hashes produced by the previous methods, these fallbacks are removed in Django 1.4.

So, if you upgrade to Django 1.4 directly from 1.2 or earlier, you may lose/invalidate certain pieces of data that have been cryptographically signed using an old method. To avoid this, use Django 1.3 first for a period of time to allow the signed data to expire naturally. The affected parts are detailed below, with 1) the consequences of ignoring this advice and 2) the amount of time you need to run Django 1.3 for the data to expire or become irrelevant.

- `contrib.sessions` data integrity check
  - Consequences: The user will be logged out, and session data will be lost.
  - Time period: Defined by `SESSION_COOKIE_AGE`.
- `contrib.auth` password reset hash
  - Consequences: Password reset links from before the upgrade will not work.
  - Time period: Defined by `PASSWORD_RESET_TIMEOUT_DAYS`.

Form-related hashes: these have a are much shorter lifetime and are relevant only for the short window where a user might fill in a form generated by the pre-upgrade Django instance and try to submit it to the upgraded Django instance:

- `contrib.comments` form security hash
  - Consequences: The user will see the validation error "Security hash failed."
  - Time period: The amount of time you expect users to take filling out comment forms.
- `FormWizard` security hash
  - Consequences: The user will see an error about the form having expired and will be sent back to the first page of the wizard, losing the data he has entered so far.
  - Time period: The amount of time you expect users to take filling out the affected forms.
- CSRF check
  - Note: This is actually a Django 1.1 fallback, not Django 1.2, and it applies only if you're upgrading from 1.1.
  - Consequences: The user will see a 403 error with any CSRF-protected POST form.
  - Time period: The amount of time you expect user to take filling out such forms.
- `contrib.auth` user password hash-upgrade sequence
  - Consequences: Each user's password will be updated to a stronger password hash when it's written to the database in 1.4. This means that if you upgrade to 1.4 and then need to downgrade to 1.3, version 1.3 won't be able to read the updated passwords.
  - Remedy: Set `PASSWORD_HASHERS` to use your original password hashing when you initially upgrade to 1.4. After you confirm your app works well with Django 1.4 and you won't have to roll back to 1.3, enable the new password hashes.

django.contrib.flatpages Starting in 1.4, the `FlatpageFallbackMiddleware` only adds a trailing slash and redirects if the resulting URL refers to an existing flatpage. For example, requesting `/notaflatpageoravalidurl` in a previous version would redirect to `/notaflatpageoravalidurl/`, which would subsequently raise a 404. Requesting `/notaflatpageoravalidurl` now will immediately raise a 404.

Also, redirects returned by flatpages are now permanent (with 301 status code), to match the behavior of `CommonMiddleware`.

Serialization of `datetime` and `time` As a consequence of time-zone support, and according to the ECMA-262 specification, we made changes to the JSON serializer:

- It includes the time zone for aware datetime objects. It raises an exception for aware time objects.
- It includes milliseconds for datetime and time objects. There is still some precision loss, because Python stores microseconds (6 digits) and JSON only supports milliseconds (3 digits). However, it's better than discarding microseconds entirely.

We changed the XML serializer to use the ISO8601 format for datetimes. The letter `T` is used to separate the date part from the time part, instead of a space. Time zone information is included in the `[+-]HH:MM` format.

Though the serializers now use these new formats when creating fixtures, they can still load fixtures that use the old format.

`supports_timezone` changed to `False` for SQLite The database feature `supports_timezone` used to be `True` for SQLite. Indeed, if you saved an aware datetime object, SQLite stored a string that included an UTC offset. However, this offset was ignored when loading the value back from the database, which could corrupt the data.

In the context of time-zone support, this flag was changed to `False`, and datetimes are now stored without time-zone information in SQLite. When `USE_TZ` is `False`, if you attempt to save an aware datetime object, Django raises an exception.

`MySQLdb`-specific exceptions The MySQL backend historically has raised `MySQLdb.OperationalError` when a query triggered an exception. We've fixed this bug, and we now raise `django.db.utils.DatabaseError` instead. If you were testing for `MySQLdb.OperationalError`, you'll need to update your `except` clauses.

Database connection's thread-locality `DatabaseWrapper` objects (i.e. the connection objects referenced by `django.db.connection` and `django.db.connections["some_alias"]`) used to be thread-local. They are now global objects in order to be potentially shared between multiple threads. While the individual connection objects are now global, the `django.db.connections` dictionary referencing those objects is still thread-local. Therefore if you just use the ORM or `DatabaseWrapper.cursor()` then the behavior is still the same as before. Note, however, that `django.db.connection` does not directly reference the default `DatabaseWrapper` object anymore and is now a proxy to access that object's attributes. If you need to access the actual `DatabaseWrapper` object, use `django.db.connections[DEFAULT_DB_ALIAS]` instead.

As part of this change, all underlying SQLite connections are now enabled for potential thread-sharing (by passing the `check_same_thread=False` attribute to pysqlite). `DatabaseWrapper` however preserves the previous behavior by disabling thread-sharing by default, so this does not affect any existing code that purely relies on the ORM or on `DatabaseWrapper.cursor()`.

Finally, while it's now possible to pass connections between threads, Django doesn't make any effort to synchronize access to the underlying backend. Concurrency behavior is defined by the underlying backend implementation. Check their documentation for details.

***COMMENTS_BANNED_USERS_GROUP* setting** Django's *comments app* has historically supported excluding the comments of a special user group, but we've never documented the feature properly and didn't enforce the exclusion in other parts of the app such as the template tags. To fix this problem, we removed the code from the feed class.

If you rely on the feature and want to restore the old behavior, use a custom comment model manager to exclude the user group, like this:

```
from django.conf import settings
from django.contrib.comments.managers import CommentManager

class BanningCommentManager(CommentManager):
    def get_query_set(self):
        qs = super(BanningCommentManager, self).get_query_set()
        if getattr(settings, 'COMMENTS_BANNED_USERS_GROUP', None):
            where = ['user_id NOT IN (SELECT user_id FROM auth_user_groups WHERE group_id = %s)']
            params = [settings.COMMENTS_BANNED_USERS_GROUP]
            qs = qs.extra(where=where, params=params)
        return qs
```

Save this model manager in your custom comment app (e.g., in `my_comments_app/managers.py`) and add it your *custom comment app model*:

```
from django.db import models
from django.contrib.comments.models import Comment

from my_comments_app.managers import BanningCommentManager

class CommentWithTitle(Comment):
    title = models.CharField(max_length=300)

    objects = BanningCommentManager()
```

For more details, see the documentation about *customizing the comments framework*.

***IGNORABLE_404_STARTS* and *IGNORABLE_404_ENDS* settings** Until Django 1.3, it was possible to exclude some URLs from Django's *404 error reporting* by adding prefixes to `IGNORABLE_404_STARTS` and suffixes to `IGNORABLE_404_ENDS`.

In Django 1.4, these two settings are superseded by `IGNORABLE_404_URLS`, which is a list of compiled regular expressions. Django won't send an email for 404 errors on URLs that match any of them.

Furthermore, the previous settings had some rather arbitrary default values:

```
IGNORABLE_404_STARTS = ('/cgi-bin/', '/_vti_bin', '/_vti_inf')
IGNORABLE_404_ENDS = ('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi',
                      'favicon.ico', '.php')
```

It's not Django's role to decide if your website has a legacy `/cgi-bin/` section or a `favicon.ico`. As a consequence, the default values of `IGNORABLE_404_URLS`, `IGNORABLE_404_STARTS` and `IGNORABLE_404_ENDS` are all now empty.

If you have customized `IGNORABLE_404_STARTS` or `IGNORABLE_404_ENDS`, or if you want to keep the old default value, you should add the following lines in your settings file:

```
import re
IGNORABLE_404_URLS = (
    # for each <prefix> in IGNORABLE_404_STARTS
    re.compile(r'^<prefix>'),
```

```
    # for each <suffix> in IGNORABLE_404_ENDS
    re.compile(r'<suffix>$'),
)
```

Don't forget to escape characters that have a special meaning in a regular expression, such as periods.

CSRF protection extended to PUT and DELETE Previously, Django's *CSRF protection* provided protection only against POST requests. Since use of PUT and DELETE methods in AJAX applications is becoming more common, we now protect all methods not defined as safe by **RFC 2616** – i.e., we exempt GET, HEAD, OPTIONS and TRACE, and we enforce protection on everything else.

If you're using PUT or DELETE methods in AJAX applications, please see the *instructions about using AJAX and CSRF*.

Password reset view now accepts `subject_template_name` The `password_reset` view in `django.contrib.auth` now accepts a `subject_template_name` parameter, which is passed to the password save form as a keyword argument. If you are using this view with a custom password reset form, then you will need to ensure your form's `save()` method accepts this keyword argument.

`django.core.template_loaders` This was an alias to `django.template.loader` since 2005, and we've removed it without emitting a warning due to the length of the deprecation. If your code still referenced this, please use `django.template.loader` instead.

`django.db.models.fields.URLField.verify_exists` This functionality has been removed due to intractable performance and security issues. Any existing usage of `verify_exists` should be removed.

`django.core.files.storage.Storage.open` The `open` method of the base Storage class used to take an obscure parameter `mixin` that allowed you to dynamically change the base classes of the returned file object. This has been removed. In the rare case you relied on the *mixin* parameter, you can easily achieve the same by overriding the *open* method, like this:

```
from django.core.files import File
from django.core.files.storage import FileSystemStorage

class Spam(File):
    """
    Spam, spam, spam, spam and spam.
    """
    def ham(self):
        return 'eggs'

class SpamStorage(FileSystemStorage):
    """
    A custom file storage backend.
    """
    def open(self, name, mode='rb'):
        return Spam(open(self.path(name), mode))
```

YAML deserializer now uses `yaml.safe_load` `yaml.load` is able to construct any Python object, which may trigger arbitrary code execution if you process a YAML document that comes from an untrusted source. This feature isn't necessary for Django's YAML deserializer, whose primary use is to load fixtures consisting of simple objects. Even though fixtures are trusted data, the YAML deserializer now uses `yaml.safe_load` for additional security.

Session cookies now have the `httponly` flag by default Session cookies now include the `httponly` attribute by default to help reduce the impact of potential XSS attacks. As a consequence of this change, session cookie data, including sessionid, is no longer accessible from JavaScript in many browsers. For strict backwards compatibility, use `SESSION_COOKIE_HTTPONLY = False` in your settings file.

The `urlize` filter no longer escapes every URL When a URL contains a `%xx` sequence, where `xx` are two hexadecimal digits, `urlize` now assumes that the URL is already escaped and doesn't apply URL escaping again. This is wrong for URLs whose unquoted form contains a `%xx` sequence, but such URLs are very unlikely to happen in the wild, because they would confuse browsers too.

`assertTemplateUsed` and `assertTemplateNotUsed` as context manager It's now possible to check whether a template was used within a block of code with `assertTemplateUsed()` and `assertTemplateNotUsed()`. And they can be used as a context manager:

```
with self.assertTemplateUsed('index.html'):
    render_to_string('index.html')
with self.assertTemplateNotUsed('base.html'):
    render_to_string('index.html')
```

See the *assertion documentation* for more.

Database connections after running the test suite The default test runner no longer restores the database connections after tests' execution. This prevents the production database from being exposed to potential threads that would still be running and attempting to create new connections.

If your code relied on connections to the production database being created after tests' execution, then you can restore the previous behavior by subclassing `DjangoTestRunner` and overriding its `teardown_databases()` method.

Output of `manage.py help` `manage.py help` now groups available commands by application. If you depended on the output of this command – if you parsed it, for example – then you'll need to update your code. To get a list of all available management commands in a script, use `manage.py help --commands` instead.

`extends` template tag Previously, the `extends` tag used a buggy method of parsing arguments, which could lead to it erroneously considering an argument as a string literal when it wasn't. It now uses `parser.compile_filter`, like other tags.

The internals of the tag aren't part of the official stable API, but in the interests of full disclosure, the `ExtendsNode.__init__` definition has changed, which may break any custom tags that use this class.

Loading some incomplete fixtures no longer works Prior to 1.4, a default value was inserted for fixture objects that were missing a specific date or datetime value when auto_now or auto_now_add was set for the field. This was something that should not have worked, and in 1.4 loading such incomplete fixtures will fail. Because fixtures are a raw import, they should explicitly specify all field values, regardless of field options on the model.

Development Server Multithreading The development server is now is multithreaded by default. Use the `--nothreading` option to disable the use of threading in the development server:

```
django-admin.py runserver --nothreading
```

Attributes disabled in markdown when safe mode set Prior to Django 1.4, attributes were included in any markdown output regardless of safe mode setting of the filter. With version > 2.1 of the Python-Markdown library, an enable_attributes option was added. When the safe argument is passed to the markdown filter, both the `safe_mode=True` and `enable_attributes=False` options are set. If using a version of the Python-Markdown library less than 2.1, a warning is issued that the output is insecure.

FormMixin get_initial returns an instance-specific dictionary In Django 1.3, the `get_initial` method of the `django.views.generic.edit.FormMixin` class was returning the class `initial` dictionary. This has been fixed to return a copy of this dictionary, so form instances can modify their initial data without messing with the class variable.

Features deprecated in 1.4

Old styles of calling `cache_page` decorator Some legacy ways of calling `cache_page()` have been deprecated. Please see the documentation for the correct way to use this decorator.

Support for PostgreSQL versions older than 8.2 Django 1.3 dropped support for PostgreSQL versions older than 8.0, and we suggested using a more recent version because of performance improvements and, more importantly, the end of upstream support periods for 8.0 and 8.1 was near (November 2010).

Django 1.4 takes that policy further and sets 8.2 as the minimum PostgreSQL version it officially supports.

Request exceptions are now always logged When we added *logging support* in Django in 1.3, the admin error email support was moved into the `django.utils.log.AdminEmailHandler`, attached to the `'django.request'` logger. In order to maintain the established behavior of error emails, the `'django.request'` logger was called only when `DEBUG` was `False`.

To increase the flexibility of error logging for requests, the `'django.request'` logger is now called regardless of the value of `DEBUG`, and the default settings file for new projects now includes a separate filter attached to `django.utils.log.AdminEmailHandler` to prevent admin error emails in `DEBUG` mode:

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse'
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

If your project was created prior to this change, your `LOGGING` setting will not include this new filter. In order to maintain backwards-compatibility, Django will detect that your `'mail_admins'` handler configuration includes no `'filters'` section and will automatically add this filter for you and issue a pending-deprecation warning. This will become a deprecation warning in Django 1.5, and in Django 1.6 the backwards-compatibility shim will be removed entirely.

The existence of any `'filters'` key under the `'mail_admins'` handler will disable this backward-compatibility shim and deprecation warning.

`django.conf.urls.defaults` Until Django 1.3, the functions `include()`, `patterns()` and `url()` plus `handler404`, `handler500` were located in a `django.conf.urls.defaults` module.

In Django 1.4, they live in `django.conf.urls`.

`django.contrib.databrowse` Databrowse has not seen active development for some time, and this does not show any sign of changing. There had been a suggestion for a GSOC project to integrate the functionality of databrowse into the admin, but no progress was made. While Databrowse has been deprecated, an enhancement of `django.contrib.admin` providing a similar feature set is still possible.

The code that powers Databrowse is licensed under the same terms as Django itself, so it's available to be adopted by an individual or group as a third-party project.

`django.core.management.setup_environ` This function temporarily modified `sys.path` in order to make the parent "project" directory importable under the old flat `startproject` layout. This function is now deprecated, as its path workarounds are no longer needed with the new `manage.py` and default project layout.

This function was never documented or part of the public API, but it was widely recommended for use in setting up a "Django environment" for a user script. These uses should be replaced by setting the `DJANGO_SETTINGS_MODULE` environment variable or using `django.conf.settings.configure()`.

`django.core.management.execute_manager` This function was previously used by `manage.py` to execute a management command. It is identical to `django.core.management.execute_from_command_line`, except that it first calls `setup_environ`, which is now deprecated. As such, `execute_manager` is also deprecated; `execute_from_command_line` can be used instead. Neither of these functions is documented as part of the public API, but a deprecation path is needed due to use in existing `manage.py` files.

`is_safe` and `needs_autoescape` attributes of template filters Two flags, `is_safe` and `needs_autoescape`, define how each template filter interacts with Django's auto-escaping behavior. They used to be attributes of the filter function:

```
@register.filter
def noop(value):
    return value
noop.is_safe = True
```

However, this technique caused some problems in combination with decorators, especially `@stringfilter`. Now, the flags are keyword arguments of `@register.filter`:

```
@register.filter(is_safe=True)
def noop(value):
    return value
```

See *filters and auto-escaping* for more information.

Wildcard expansion of application names in *INSTALLED_APPS* Until Django 1.3, `INSTALLED_APPS` accepted wildcards in application names, like `django.contrib.*`. The expansion was performed by a filesystem-based implementation of `from <package> import *`. Unfortunately, this can't be done reliably.

This behavior was never documented. Since it is un-pythonic and not obviously useful, it was removed in Django 1.4. If you relied on it, you must edit your settings file to list all your applications explicitly.

`HttpRequest.raw_post_data` renamed to `HttpRequest.body` This attribute was confusingly named `HttpRequest.raw_post_data`, but it actually provided the body of the HTTP request. It's been renamed to `HttpRequest.body`, and `HttpRequest.raw_post_data` has been deprecated.

`django.contrib.sitemaps` bug fix with potential performance implications In previous versions, `Paginator` objects used in sitemap classes were cached, which could result in stale site maps. We've removed the caching, so each request to a site map now creates a new Paginator object and calls the `items()` method of the `Sitemap` subclass. Depending on what your `items()` method is doing, this may have a negative performance impact. To mitigate the performance impact, consider using the *caching framework* within your `Sitemap` subclass.

Versions of Python-Markdown earlier than 2.1 Versions of Python-Markdown earlier than 2.1 do not support the option to disable attributes. As a security issue, earlier versions of this library will not be supported by the markup contrib app in 1.5 under an accelerated deprecation timeline.

9.1.3 1.3 release

Django 1.3.1 release notes

September 9, 2011

Welcome to Django 1.3.1!

This is the first security release in the Django 1.3 series, fixing several security issues in Django 1.3. Django 1.3.1 is a recommended upgrade for all users of Django 1.3.

For a full list of issues addressed in this release, see the security advisory.

Django 1.3 release notes

March 23, 2011

Welcome to Django 1.3!

Nearly a year in the making, Django 1.3 includes quite a few new features and plenty of bug fixes and improvements to existing features. These release notes cover the new features in 1.3, as well as some backwards-incompatible changes you'll want to be aware of when upgrading from Django 1.2 or older versions.

Overview

Django 1.3's focus has mostly been on resolving smaller, long-standing feature requests, but that hasn't prevented a few fairly significant new features from landing, including:

- A framework for writing class-based views.
- Built-in support for using Python's logging facilities.
- Contrib support for easy handling of static files.
- Django's testing framework now supports (and ships with a copy of) the unittest2 library.

There's plenty more, of course; see the coverage of new features below for a full rundown and details.

Wherever possible, of course, new features are introduced in a backwards-compatible manner per *our API stability policy* policy. As a result of this policy, Django 1.3 begins the deprecation process for some features.

Some changes, unfortunately, are genuinely backwards-incompatible; in most cases these are due to security issues or bugs which simply couldn't be fixed any other way. Django 1.3 includes a few of these, and descriptions of them – along with the (minor) modifications you'll need to make to handle them – are documented in the list of backwards-incompatible changes below.

Python compatibility

The release of Django 1.2 was notable for having the first shift in Django's Python compatibility policy; prior to Django 1.2, Django supported any 2.x version of Python from 2.3 up. As of Django 1.2, the minimum requirement was raised to Python 2.4.

Django 1.3 continues to support Python 2.4, but will be the final Django release series to do so; beginning with Django 1.4, the minimum supported Python version will be 2.5. A document outlining our full timeline for deprecating Python 2.x and moving to Python 3.x will be published shortly after the release of Django 1.3.

What's new in Django 1.3

Class-based views Django 1.3 adds a framework that allows you to use a class as a view. This means you can compose a view out of a collection of methods that can be subclassed and overridden to provide common views of data without having to write too much code.

Analogs of all the old function-based generic views have been provided, along with a completely generic view base class that can be used as the basis for reusable applications that can be easily extended.

See *the documentation on class-based generic views* for more details. There is also a document to help you convert your function-based generic views to class-based views.

Logging Django 1.3 adds framework-level support for Python's `logging` module. This means you can now easily configure and control logging as part of your Django project. A number of logging handlers and logging calls have been added to Django's own code as well – most notably, the error emails sent on a HTTP 500 server error are now handled as a logging activity. See *the documentation on Django's logging interface* for more details.

Extended static files handling Django 1.3 ships with a new contrib app – `django.contrib.staticfiles` – to help developers handle the static media files (images, CSS, Javascript, etc.) that are needed to render a complete web page.

In previous versions of Django, it was common to place static assets in `MEDIA_ROOT` along with user-uploaded files, and serve them both at `MEDIA_URL`. Part of the purpose of introducing the `staticfiles` app is to make it easier to keep static files separate from user-uploaded files. Static assets should now go in `static/` subdirectories of your apps or in other static assets directories listed in `STATICFILES_DIRS`, and will be served at `STATIC_URL`.

See the *reference documentation of the app* for more details or learn how to *manage static files*.

unittest2 support Python 2.7 introduced some major changes to the `unittest` library, adding some extremely useful features. To ensure that every Django project can benefit from these new features, Django ships with a copy of unittest2, a copy of the Python 2.7 unittest library, backported for Python 2.4 compatibility.

To access this library, Django provides the `django.utils.unittest` module alias. If you are using Python 2.7, or you have installed `unittest2` locally, Django will map the alias to the installed version of the unittest library. Otherwise, Django will use its own bundled version of unittest2.

To take advantage of this alias, simply use:

```
from django.utils import unittest
```

wherever you would have historically used:

```
import unittest
```

If you want to continue to use the base unittest library, you can – you just won't get any of the nice new unittest2 features.

Transaction context managers Users of Python 2.5 and above may now use *transaction management functions* as context managers. For example:

```
with transaction.autocommit():
    # ...
```

For more information, see *Controlling transaction management in views*.

Configurable delete-cascade `ForeignKey` and `OneToOneField` now accept an `on_delete` argument to customize behavior when the referenced object is deleted. Previously, deletes were always cascaded; available alternatives now include set null, set default, set to any value, protect, or do nothing.

For more information, see the `on_delete` documentation.

Contextual markers and comments for translatable strings For translation strings with ambiguous meaning, you can now use the `pgettext` function to specify the context of the string.

And if you just want to add some information for translators, you can also add special translator comments in the source.

For more information, see *Contextual markers* and *Comments for translators*.

Improvements to built-in template tags A number of improvements have been made to Django's built-in template tags:

- The `include` tag now accepts a `with` option, allowing you to specify context variables to the included template
- The `include` tag now accepts an `only` option, allowing you to exclude the current context from the included context
- The `with` tag now allows you to define multiple context variables in a single `with` block.
- The `load` tag now accepts a `from` argument, allowing you to load a single tag or filter from a library.

TemplateResponse It can sometimes be beneficial to allow decorators or middleware to modify a response *after* it has been constructed by the view. For example, you may want to change the template that is used, or put additional data into the context.

However, you can't (easily) modify the content of a basic `HttpResponse` after it has been constructed. To overcome this limitation, Django 1.3 adds a new `TemplateResponse` class. Unlike basic `HttpResponse` objects, `TemplateResponse` objects retain the details of the template and context that was provided by the view to compute the response. The final output of the response is not computed until it is needed, later in the response process.

For more details, see the *documentation* on the `TemplateResponse` class.

Caching changes Django 1.3 sees the introduction of several improvements to the Django's caching infrastructure.

Firstly, Django now supports multiple named caches. In the same way that Django 1.2 introduced support for multiple database connections, Django 1.3 allows you to use the new `CACHES` setting to define multiple named cache connections.

Secondly, *versioning*, *site-wide prefixing* and *transformation* have been added to the cache API.

Thirdly, *cache key creation* has been updated to take the request query string into account on `GET` requests.

Finally, support for pylibmc has been added to the memcached cache backend.

For more details, see the *documentation on caching in Django*.

Permissions for inactive users If you provide a custom auth backend with `supports_inactive_user` set to `True`, an inactive `User` instance will check the backend for permissions. This is useful for further centralizing the permission handling. See the *authentication docs* for more details.

GeoDjango The GeoDjango test suite is now included when *running the Django test suite* with `runtests.py` when using *spatial database backends*.

`MEDIA_URL` and `STATIC_URL` must end in a slash Previously, the `MEDIA_URL` setting only required a trailing slash if it contained a suffix beyond the domain name.

A trailing slash is now *required* for `MEDIA_URL` and the new `STATIC_URL` setting as long as it is not blank. This ensures there is a consistent way to combine paths in templates.

Project settings which provide either of both settings without a trailing slash will now raise a `PendingDeprecationWarning`.

In Django 1.4 this same condition will raise `DeprecationWarning`, and in Django 1.5 will raise an `ImproperlyConfigured` exception.

Everything else Django *1.1* and *1.2* added lots of big ticket items to Django, like multiple-database support, model validation, and a session-based messages framework. However, this focus on big features came at the cost of lots of smaller features.

To compensate for this, the focus of the Django 1.3 development process has been on adding lots of smaller, long standing feature requests. These include:

- Improved tools for accessing and manipulating the current `Site` object in *the sites framework*.
- A `RequestFactory` for mocking requests in tests.
- A new test assertion – `assertNumQueries()` – making it easier to test the database activity associated with a view.
- Support for lookups spanning relations in admin's `list_filter`.
- Support for HTTPOnly cookies.
- `mail_admins()` and `mail_managers()` now support easily attaching HTML content to messages.
- `EmailMessage` now supports CC's.
- Error emails now include more of the detail and formatting of the debug server error page.
- `simple_tag()` now accepts a `takes_context` argument, making it easier to write simple template tags that require access to template context.

- A new `render()` shortcut – an alternative to `render_to_response()` providing a `RequestContext` by default.
- Support for combining *F() expressions* with timedelta values when retrieving or updating database values.

Backwards-incompatible changes in 1.3

CSRF validation now applies to AJAX requests Prior to Django 1.2.5, Django's CSRF-prevention system exempted AJAX requests from CSRF verification; due to security issues reported to us, however, *all* requests are now subjected to CSRF verification. Consult *the Django CSRF documentation* for details on how to handle CSRF verification in AJAX requests.

Restricted filters in admin interface Prior to Django 1.2.5, the Django administrative interface allowed filtering on any model field or relation – not just those specified in `list_filter` – via query string manipulation. Due to security issues reported to us, however, query string lookup arguments in the admin must be for fields or relations specified in `list_filter` or `date_hierarchy`.

Deleting a model doesn't delete associated files In earlier Django versions, when a model instance containing a `FileField` was deleted, `FileField` took it upon itself to also delete the file from the backend storage. This opened the door to several data-loss scenarios, including rolled-back transactions and fields on different models referencing the same file. In Django 1.3, when a model is deleted the `FileField`'s `delete()` method won't be called. If you need cleanup of orphaned files, you'll need to handle it yourself (for instance, with a custom management command that can be run manually or scheduled to run periodically via e.g. cron).

PasswordInput default rendering behavior The `PasswordInput` form widget, intended for use with form fields which represent passwords, accepts a boolean keyword argument `render_value` indicating whether to send its data back to the browser when displaying a submitted form with errors. Prior to Django 1.3, this argument defaulted to `True`, meaning that the submitted password would be sent back to the browser as part of the form. Developers who wished to add a bit of additional security by excluding that value from the redisplayed form could instantiate a `PasswordInput` passing `render_value=False`.

Due to the sensitive nature of passwords, however, Django 1.3 takes this step automatically; the default value of `render_value` is now `False`, and developers who want the password value returned to the browser on a submission with errors (the previous behavior) must now explicitly indicate this. For example:

```
class LoginForm(forms.Form):
    username = forms.CharField(max_length=100)
    password = forms.CharField(widget=forms.PasswordInput(render_value=True))
```

Clearable default widget for FileField Django 1.3 now includes a `ClearableFileInput` form widget in addition to `FileInput`. `ClearableFileInput` renders with a checkbox to clear the field's value (if the field has a value and is not required); `FileInput` provided no means for clearing an existing file from a `FileField`.

`ClearableFileInput` is now the default widget for a `FileField`, so existing forms including `FileField` without assigning a custom widget will need to account for the possible extra checkbox in the rendered form output.

To return to the previous rendering (without the ability to clear the `FileField`), use the `FileInput` widget in place of `ClearableFileInput`. For instance, in a `ModelForm` for a hypothetical `Document` model with a `FileField` named `document`:

```
from django import forms
from myapp.models import Document
```

```
class DocumentForm(forms.ModelForm):
    class Meta:
        model = Document
        widgets = {'document': forms.FileInput}
```

New index on database session table Prior to Django 1.3, the database table used by the database backend for the *sessions* app had no index on the `expire_date` column. As a result, date-based queries on the session table – such as the query that is needed to purge old sessions – would be very slow if there were lots of sessions.

If you have an existing project that is using the database session backend, you don't have to do anything to accommodate this change. However, you may get a significant performance boost if you manually add the new index to the session table. The SQL that will add the index can be found by running the `sqlindexes` admin command:

```
python manage.py sqlindexes sessions
```

No more naughty words Django has historically provided (and enforced) a list of profanities. The *comments app* has enforced this list of profanities, preventing people from submitting comments that contained one of those profanities.

Unfortunately, the technique used to implement this profanities list was woefully naive, and prone to the Scunthorpe problem. Improving the built-in filter to fix this problem would require significant effort, and since natural language processing isn't the normal domain of a web framework, we have "fixed" the problem by making the list of prohibited words an empty list.

If you want to restore the old behavior, simply put a `PROFANITIES_LIST` setting in your settings file that includes the words that you want to prohibit (see the commit that implemented this change if you want to see the list of words that was historically prohibited). However, if avoiding profanities is important to you, you would be well advised to seek out a better, less naive approach to the problem.

Localflavor changes Django 1.3 introduces the following backwards-incompatible changes to local flavors:

- Canada (ca) – The province "Newfoundland and Labrador" has had its province code updated to "NL", rather than the older "NF". In addition, the Yukon Territory has had its province code corrected to "YT", instead of "YK".
- Indonesia (id) – The province "Nanggroe Aceh Darussalam (NAD)" has been removed from the province list in favor of the new official designation "Aceh (ACE)".
- United States of America (us) – The list of "states" used by `USStateField` has expanded to include Armed Forces postal codes. This is backwards-incompatible if you were relying on `USStateField` not including them.

FormSet updates In Django 1.3 `FormSet` creation behavior is modified slightly. Historically the class didn't make a distinction between not being passed data and being passed empty dictionary. This was inconsistent with behavior in other parts of the framework. Starting with 1.3 if you pass in empty dictionary the `FormSet` will raise a `ValidationError`.

For example with a `FormSet`:

```
>>> class ArticleForm(Form):
...     title = CharField()
...     pub_date = DateField()
>>> ArticleFormSet = formset_factory(ArticleForm)
```

the following code will raise a `ValidationError`:

```
>>> ArticleFormSet({})
Traceback (most recent call last):
...
ValidationError: [u'ManagementForm data is missing or has been tampered with']
```

if you need to instantiate an empty `FormSet`, don't pass in the data or use `None`:

```
>>> formset = ArticleFormSet()
>>> formset = ArticleFormSet(data=None)
```

Callables in templates Previously, a callable in a template would only be called automatically as part of the variable resolution process if it was retrieved via attribute lookup. This was an inconsistency that could result in confusing and unhelpful behavior:

```
>>> Template("{{ user.get_full_name }}").render(Context({'user': user}))
u'Joe Bloggs'
>>> Template("{{ full_name }}").render(Context({'full_name': user.get_full_name}))
u'&lt;bound method User.get_full_name of &lt;...
```

This has been resolved in Django 1.3 - the result in both cases will be `u'Joe Bloggs'`. Although the previous behavior was not useful for a template language designed for web designers, and was never deliberately supported, it is possible that some templates may be broken by this change.

Use of custom SQL to load initial data in tests Django provides a custom SQL hooks as a way to inject hand-crafted SQL into the database synchronization process. One of the possible uses for this custom SQL is to insert data into your database. If your custom SQL contains `INSERT` statements, those insertions will be performed every time your database is synchronized. This includes the synchronization of any test databases that are created when you run a test suite.

However, in the process of testing the Django 1.3, it was discovered that this feature has never completely worked as advertised. When using database backends that don't support transactions, or when using a TransactionTestCase, data that has been inserted using custom SQL will not be visible during the testing process.

Unfortunately, there was no way to rectify this problem without introducing a backwards incompatibility. Rather than leave SQL-inserted initial data in an uncertain state, Django now enforces the policy that data inserted by custom SQL will *not* be visible during testing.

This change only affects the testing process. You can still use custom SQL to load data into your production database as part of the syncdb process. If you require data to exist during test conditions, you should either insert it using *test fixtures*, or using the `setUp()` method of your test case.

Changed priority of translation loading Work has been done to simplify, rationalize and properly document the algorithm used by Django at runtime to build translations from the different translations found on disk, namely:

For translatable literals found in Python code and templates (`'django'` gettext domain):

- Priorities of translations included with applications listed in the `INSTALLED_APPS` setting were changed. To provide a behavior consistent with other parts of Django that also use such setting (templates, etc.) now, when building the translation that will be made available, the apps listed first have higher precedence than the ones listed later.
- Now it is possible to override the translations shipped with applications by using the `LOCALE_PATHS` setting whose translations have now higher precedence than the translations of `INSTALLED_APPS` applications. The relative priority among the values listed in this setting has also been modified so the paths listed first have higher precedence than the ones listed later.

- The `locale` subdirectory of the directory containing the settings, that usually coincides with and is know as the *project directory* is being deprecated in this release as a source of translations. (the precedence of these translations is intermediate between applications and `LOCALE_PATHS` translations). See the corresponding deprecated features section of this document.

For translatable literals found in Javascript code (`'djangojs'` gettext domain):

- Similarly to the `'django'` domain translations: Overriding of translations shipped with applications by using the `LOCALE_PATHS` setting is now possible for this domain too. These translations have higher precedence than the translations of Python packages passed to the *javascript_catalog view*. Paths listed first have higher precedence than the ones listed later.
- Translations under the `locale` subdirectory of the *project directory* have never been taken in account for JavaScript translations and remain in the same situation considering the deprecation of such location.

Transaction management When using managed transactions – that is, anything but the default autocommit mode – it is important when a transaction is marked as "dirty". Dirty transactions are committed by the `commit_on_success()` decorator or the `TransactionMiddleware`, and `commit_manually()` forces them to be closed explicitly; clean transactions "get a pass", which means they are usually rolled back at the end of a request when the connection is closed.

Until Django 1.3, transactions were only marked dirty when Django was aware of a modifying operation performed in them; that is, either some model was saved, some bulk update or delete was performed, or the user explicitly called `transaction.set_dirty()`. In Django 1.3, a transaction is marked dirty when *any* database operation is performed.

As a result of this change, you no longer need to set a transaction dirty explicitly when you execute raw SQL or use a data-modifying `SELECT`. However, you *do* need to explicitly close any read-only transactions that are being managed using `commit_manually()`. For example:

```
@transaction.commit_manually
def my_view(request, name):
    obj = get_object_or_404(MyObject, name__iexact=name)
    return render_to_response('template', {'object':obj})
```

Prior to Django 1.3, this would work without error. However, under Django 1.3, this will raise a `TransactionManagementError` because the read operation that retrieves the `MyObject` instance leaves the transaction in a dirty state.

No password reset for inactive users Prior to Django 1.3, inactive users were able to request a password reset email and reset their password. In Django 1.3 inactive users will receive the same message as a nonexistent account.

Features deprecated in 1.3

Django 1.3 deprecates some features from earlier releases. These features are still supported, but will be gradually phased out over the next few release cycles.

Code taking advantage of any of the features below will raise a `PendingDeprecationWarning` in Django 1.3. This warning will be silent by default, but may be turned on using Python's `warnings` module, or by running Python with a `-Wd` or *-Wall* flag.

In Django 1.4, these warnings will become a `DeprecationWarning`, which is *not* silent. In Django 1.5 support for these features will be removed entirely.

See Also:

For more details, see the documentation *Django's release process* and our *deprecation timeline*.

mod_python support The mod_python library has not had a release since 2007 or a commit since 2008. The Apache Foundation board voted to remove mod_python from the set of active projects in its version control repositories, and its lead developer has shifted all of his efforts toward the lighter, slimmer, more stable, and more flexible mod_wsgi backend.

If you are currently using the mod_python request handler, you should redeploy your Django projects using another request handler. *mod_wsgi* is the request handler recommended by the Django project, but *FastCGI* is also supported. Support for mod_python deployment will be removed in Django 1.5.

Function-based generic views As a result of the introduction of class-based generic views, the function-based generic views provided by Django have been deprecated. The following modules and the views they contain have been deprecated:

- django.views.generic.create_update
- django.views.generic.date_based
- django.views.generic.list_detail
- django.views.generic.simple

Test client response template attribute Django's *test client* returns Response objects annotated with extra testing information. In Django versions prior to 1.3, this included a template attribute containing information about templates rendered in generating the response: either None, a single Template object, or a list of Template objects. This inconsistency in return values (sometimes a list, sometimes not) made the attribute difficult to work with.

In Django 1.3 the template attribute is deprecated in favor of a new templates attribute, which is always a list, even if it has only a single element or no elements.

DjangoTestRunner As a result of the introduction of support for unittest2, the features of django.test.simple.DjangoTestRunner (including fail-fast and Ctrl-C test termination) have been made redundant. In view of this redundancy, DjangoTestRunner has been turned into an empty placeholder class, and will be removed entirely in Django 1.5.

Changes to url and ssi Most template tags will allow you to pass in either constants or variables as arguments – for example:

```
{% extends "base.html" %}
```

allows you to specify a base template as a constant, but if you have a context variable templ that contains the value base.html:

```
{% extends templ %}
```

is also legal.

However, due to an accident of history, the url and ssi are different. These tags use the second, quoteless syntax, but interpret the argument as a constant. This means it isn't possible to use a context variable as the target of a url and ssi tag.

Django 1.3 marks the start of the process to correct this historical accident. Django 1.3 adds a new template library – future – that provides alternate implementations of the url and ssi template tags. This future library implement behavior that makes the handling of the first argument consistent with the handling of all other variables. So, an existing template that contains:

```
{% url sample %}
```

should be replaced with:

```
{% load url from future %}
{% url 'sample' %}
```

The tags implementing the old behavior have been deprecated, and in Django 1.5, the old behavior will be replaced with the new behavior. To ensure compatibility with future versions of Django, existing templates should be modified to use the new `future` libraries and syntax.

Changes to the login methods of the admin In previous version the admin app defined login methods in multiple locations and ignored the almost identical implementation in the already used auth app. A side effect of this duplication was the missing adoption of the changes made in r12634 to support a broader set of characters for usernames.

This release refactors the admin's login mechanism to use a subclass of the `AuthenticationForm` instead of a manual form validation. The previously undocumented method `'django.contrib.admin.sites.AdminSite.display_login_form'` has been removed in favor of a new `login_form` attribute.

`reset` and `sqlreset` management commands Those commands have been deprecated. The `flush` and `sqlflush` commands can be used to delete everything. You can also use ALTER TABLE or DROP TABLE statements manually.

GeoDjango

- The function-based `TEST_RUNNER` previously used to execute the GeoDjango test suite, `django.contrib.gis.tests.run_gis_tests()`, was deprecated for the class-based runner, `django.contrib.gis.tests.GeoDjangoTestSuiteRunner`.
- Previously, calling `transform()` would silently do nothing when GDAL wasn't available. Now, a `GEOSException` is properly raised to indicate possible faulty application code. A warning is now raised if `transform()` is called when the SRID of the geometry is less than 0 or `None`.

`CZBirthNumberField.clean` Previously this field's `clean()` method accepted a second, gender, argument which allowed stronger validation checks to be made, however since this argument could never actually be passed from the Django form machinery it is now pending deprecation.

`CompatCookie` Previously, `django.http` exposed an undocumented `CompatCookie` class, which was a bugfix wrapper around the standard library `SimpleCookie`. As the fixes are moving upstream, this is now deprecated - you should use `from django.http import SimpleCookie` instead.

Loading of *project-level* translations This release of Django starts the deprecation process for inclusion of translations located under the so-called *project path* in the translation building process performed at runtime. The `LOCALE_PATHS` setting can be used for the same task by adding the filesystem path to a `locale` directory containing project-level translations to the value of that setting.

Rationale for this decision:

- The *project path* has always been a loosely defined concept (actually, the directory used for locating project-level translations is the directory containing the settings module) and there has been a shift in other parts of the framework to stop using it as a reference for location of assets at runtime.
- Detection of the `locale` subdirectory tends to fail when the deployment scenario is more complex than the basic one. e.g. it fails when the settings module is a directory (ticket #10765).

- There are potential strange development- and deployment-time problems like the fact that the `project_dir/locale/` subdir can generate spurious error messages when the project directory is added to the Python path (`manage.py runserver` does this) and then it clashes with the equally named standard library module, this is a typical warning message:

  ```
  /usr/lib/python2.6/gettext.py:49: ImportWarning: Not importing directory '/path/to/project/local
  import locale, copy, os, re, struct, sys
  ```

- This location wasn't included in the translation building process for JavaScript literals. This deprecation removes such inconsistency.

`PermWrapper` moved to `django.contrib.auth.context_processors` In Django 1.2, we began the process of changing the location of the `auth` context processor from `django.core.context_processors` to `django.contrib.auth.context_processors`. However, the `PermWrapper` support class was mistakenly omitted from that migration. In Django 1.3, the `PermWrapper` class has also been moved to `django.contrib.auth.context_processors`, along with the `PermLookupDict` support class. The new classes are functionally identical to their old versions; only the module location has changed.

Removal of `XMLField` When Django was first released, Django included an `XMLField` that performed automatic XML validation for any field input. However, this validation function hasn't been performed since the introduction of `newforms`, prior to the 1.0 release. As a result, `XMLField` as currently implemented is functionally indistinguishable from a simple `TextField`.

For this reason, Django 1.3 has fast-tracked the deprecation of `XMLField` – instead of a two-release deprecation, `XMLField` will be removed entirely in Django 1.4.

It's easy to update your code to accommodate this change – just replace all uses of `XMLField` with `TextField`, and remove the `schema_path` keyword argument (if it is specified).

9.1.4 1.2 release

Django 1.2.7 release notes

September 10, 2011

Welcome to Django 1.2.7!

This is the seventh bugfix/security release in the Django 1.2 series. It replaces Django 1.2.6 due to problems with the 1.2.6 release tarball. Django 1.2.7 is a recommended upgrade for all users of any Django release in the 1.2.X series.

For more information, see the release advisory.

Django 1.2.6 release notes

September 9, 2011

Welcome to Django 1.2.6!

This is the sixth bugfix/security release in the Django 1.2 series, fixing several security issues present in Django 1.2.5. Django 1.2.6 is a recommended upgrade for all users of any Django release in the 1.2.X series.

For a full list of issues addressed in this release, see the security advisory.

Django 1.2.5 release notes

Welcome to Django 1.2.5!

This is the fifth "bugfix" release in the Django 1.2 series, improving the stability and performance of the Django 1.2 codebase.

With four exceptions, Django 1.2.5 maintains backwards compatibility with Django 1.2.4. It also contains a number of fixes and other improvements. Django 1.2.5 is a recommended upgrade for any development or deployment currently using or targeting Django 1.2.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.2 branch, see the *Django 1.2 release notes*.

Backwards incompatible changes

CSRF exception for AJAX requests Django includes a CSRF-protection mechanism, which makes use of a token inserted into outgoing forms. Middleware then checks for the token's presence on form submission, and validates it.

Prior to Django 1.2.5, our CSRF protection made an exception for AJAX requests, on the following basis:

- Many AJAX toolkits add an X-Requested-With header when using XMLHttpRequest.
- Browsers have strict same-origin policies regarding XMLHttpRequest.
- In the context of a browser, the only way that a custom header of this nature can be added is with XMLHttpRequest.

Therefore, for ease of use, we did not apply CSRF checks to requests that appeared to be AJAX on the basis of the X-Requested-With header. The Ruby on Rails web framework had a similar exemption.

Recently, engineers at Google made members of the Ruby on Rails development team aware of a combination of browser plugins and redirects which can allow an attacker to provide custom HTTP headers on a request to any website. This can allow a forged request to appear to be an AJAX request, thereby defeating CSRF protection which trusts the same-origin nature of AJAX requests.

Michael Koziarski of the Rails team brought this to our attention, and we were able to produce a proof-of-concept demonstrating the same vulnerability in Django's CSRF handling.

To remedy this, Django will now apply full CSRF validation to all requests, regardless of apparent AJAX origin. This is technically backwards-incompatible, but the security risks have been judged to outweigh the compatibility concerns in this case.

Additionally, Django will now accept the CSRF token in the custom HTTP header X-CSRFTOKEN, as well as in the form submission itself, for ease of use with popular JavaScript toolkits which allow insertion of custom headers into all AJAX requests.

Please see the *CSRF docs for example jQuery code* that demonstrates this technique, ensuring that you are looking at the documentation for your version of Django, as the exact code necessary is different for some older versions of Django.

FileField no longer deletes files In earlier Django versions, when a model instance containing a `FileField` was deleted, `FileField` took it upon itself to also delete the file from the backend storage. This opened the door to several potentially serious data-loss scenarios, including rolled-back transactions and fields on different models referencing the same file. In Django 1.2.5, `FileField` will never delete files from the backend storage. If you need cleanup of orphaned files, you'll need to handle it yourself (for instance, with a custom management command that can be run manually or scheduled to run periodically via e.g. cron).

Use of custom SQL to load initial data in tests Django provides a custom SQL hooks as a way to inject hand-crafted SQL into the database synchronization process. One of the possible uses for this custom SQL is to insert data into your database. If your custom SQL contains `INSERT` statements, those insertions will be performed every time your database is synchronized. This includes the synchronization of any test databases that are created when you run a test suite.

However, in the process of testing the Django 1.3, it was discovered that this feature has never completely worked as advertised. When using database backends that don't support transactions, or when using a TransactionTestCase, data that has been inserted using custom SQL will not be visible during the testing process.

Unfortunately, there was no way to rectify this problem without introducing a backwards incompatibility. Rather than leave SQL-inserted initial data in an uncertain state, Django now enforces the policy that data inserted by custom SQL will *not* be visible during testing.

This change only affects the testing process. You can still use custom SQL to load data into your production database as part of the syncdb process. If you require data to exist during test conditions, you should either insert it using *test fixtures*, or using the `setUp()` method of your test case.

ModelAdmin.lookup_allowed signature changed Django 1.2.4 introduced a method `lookup_allowed` on `ModelAdmin`, to cope with a security issue (changeset [15033]). Although this method was never documented, it seems some people have overridden `lookup_allowed`, especially to cope with regressions introduced by that changeset. While the method is still undocumented and not marked as stable, it may be helpful to know that the signature of this function has changed.

Django 1.2.4 release notes

Welcome to Django 1.2.4!

This is the fourth "bugfix" release in the Django 1.2 series, improving the stability and performance of the Django 1.2 codebase.

With one exception, Django 1.2.4 maintains backwards compatibility with Django 1.2.3. It also contains a number of fixes and other improvements. Django 1.2.4 is a recommended upgrade for any development or deployment currently using or targeting Django 1.2.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.2 branch, see the *Django 1.2 release notes*.

Backwards incompatible changes

Restricted filters in admin interface The Django administrative interface, django.contrib.admin, supports filtering of displayed lists of objects by fields on the corresponding models, including across database-level relationships. This is implemented by passing lookup arguments in the querystring portion of the URL, and options on the ModelAdmin class allow developers to specify particular fields or relationships which will generate automatic links for filtering.

One historically-undocumented and -unofficially-supported feature has been the ability for a user with sufficient knowledge of a model's structure and the format of these lookup arguments to invent useful new filters on the fly by manipulating the querystring.

However, it has been demonstrated that this can be abused to gain access to information outside of an admin user's permissions; for example, an attacker with access to the admin and sufficient knowledge of model structure and relations could construct query strings which – with repeated use of regular-expression lookups supported by the Django database API – expose sensitive information such as users' password hashes.

To remedy this, django.contrib.admin will now validate that querystring lookup arguments either specify only fields on the model being viewed, or cross relations which have been explicitly whitelisted by the application developer

using the pre-existing mechanism mentioned above. This is backwards-incompatible for any users relying on the prior ability to insert arbitrary lookups.

One new feature

Ordinarily, a point release would not include new features, but in the case of Django 1.2.4, we have made an exception to this rule.

One of the bugs fixed in Django 1.2.4 involves a set of circumstances whereby a running a test suite on a multiple database configuration could cause the original source database (i.e., the actual production database) to be dropped, causing catastrophic loss of data. In order to provide a fix for this problem, it was necessary to introduce a new setting – `TEST_DEPENDENCIES` – that allows you to define any creation order dependencies in your database configuration.

Most users – even users with multiple-database configurations – need not be concerned about the data loss bug, or the manual configuration of `TEST_DEPENDENCIES`. See the original problem report documentation on *controlling the creation order of test databases* for details.

GeoDjango

The function-based `TEST_RUNNER` previously used to execute the GeoDjango test suite, `django.contrib.gis.tests.run_gis_tests()`, was finally deprecated in favor of a class-based test runner, `django.contrib.gis.tests.GeoDjangoTestSuiteRunner`, added in this release.

In addition, the GeoDjango test suite is now included when *running the Django test suite* with `runtests.py` and using *spatial database backends*.

Django 1.2.3 release notes

Django 1.2.3 fixed a couple of release problems in the 1.2.2 release and was released two days after 1.2.2.

This release corrects the following problems:

- The patch applied for the security issue covered in Django 1.2.2 caused issues with non-ASCII responses using CSRF tokens.
- The patch also caused issues with some forms, most notably the user-editing forms in the Django administrative interface.
- The packaging manifest did not contain the full list of required files.

Django 1.2.2 release notes

Welcome to Django 1.2.2!

This is the second "bugfix" release in the Django 1.2 series, improving the stability and performance of the Django 1.2 codebase.

Django 1.2.2 maintains backwards compatibility with Django 1.2.1, but contain a number of fixes and other improvements. Django 1.2.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.2.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.2 branch, see the *Django 1.2 release notes*.

One new feature

Ordinarily, a point release would not include new features, but in the case of Django 1.2.2, we have made an exception to this rule.

In order to test a bug fix that forms part of the 1.2.2 release, it was necessary to add a feature – the `enforce_csrf_checks` flag – to the `test client`. This flag forces the test client to perform full CSRF checks on forms. The default behavior of the test client hasn't changed, but if you want to do CSRF checks with the test client, it is now possible to do so.

Django 1.2.1 release notes

Django 1.2.1 was released almost immediately after 1.2.0 to correct two small bugs: one was in the documentation packaging script, the other was a bug that affected datetime form field widgets when localisation was enabled.

Django 1.2 release notes

May 17, 2010.

Welcome to Django 1.2!

Nearly a year in the making, Django 1.2 packs an impressive list of new features and lots of bug fixes. These release notes cover the new features, as well as important changes you'll want to be aware of when upgrading from Django 1.1 or older versions.

Overview

Django 1.2 introduces several large, important new features, including:

- Support for multiple database connections in a single Django instance.
- Model validation inspired by Django's form validation.
- Vastly improved protection against Cross-Site Request Forgery (CSRF).
- A new user "messages" framework with support for cookie- and session-based message for both anonymous and authenticated users.
- Hooks for object-level permissions, permissions for anonymous users, and more flexible username requirements.
- Customization of email sending via email backends.
- New *"smart" if template tag* which supports comparison operators.

These are just the highlights; full details and a complete list of features may be found below.

See Also:

Django Advent covered the release of Django 1.2 with a series of articles and tutorials that cover some of the new features in depth.

Wherever possible these features have been introduced in a backwards-compatible manner per *our API stability policy* policy.

However, a handful of features *have* changed in ways that, for some users, will be backwards-incompatible. The big changes are:

- Support for Python 2.3 has been dropped. See the full notes below.

- The new CSRF protection framework is not backwards-compatible with the old system. Users of the old system will not be affected until the old system is removed in Django 1.4.

  However, upgrading to the new CSRF protection framework requires a few important backwards-incompatible changes, detailed in CSRF Protection, below.

- Authors of custom `Field` subclasses should be aware that a number of methods have had a change in prototype, detailed under get_db_prep_*() methods on Field, below.

- The internals of template tags have changed somewhat; authors of custom template tags that need to store state (e.g. custom control flow tags) should ensure that their code follows the new rules for stateful template tags

- The `user_passes_test()`, `login_required()`, and `permission_required()`, decorators from `django.contrib.auth` only apply to functions and no longer work on methods. There's a simple one-line fix detailed below.

Again, these are just the big features that will affect the most users. Users upgrading from previous versions of Django are heavily encouraged to consult the complete list of *backwards-incompatible changes* and the list of *deprecated features*.

Python compatibility

While not a new feature, it's important to note that Django 1.2 introduces the first shift in our Python compatibility policy since Django's initial public debut. Previous Django releases were tested and supported on 2.x Python versions from 2.3 up; Django 1.2, however, drops official support for Python 2.3. As such, the minimum Python version required for Django is now 2.4, and Django is tested and supported on Python 2.4, 2.5 and 2.6, and will be supported on the as-yet-unreleased Python 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.4 or newer as their default version. If you're still using Python 2.3, however, you'll need to stick to Django 1.1 until you can upgrade; per *our support policy*, Django 1.1 will continue to receive security support until the release of Django 1.3.

A roadmap for Django's overall 2.x Python support, and eventual transition to Python 3.x, is currently being developed, and will be announced prior to the release of Django 1.3.

What's new in Django 1.2

Support for multiple databases Django 1.2 adds the ability to use *more than one database* in your Django project. Queries can be issued at a specific database with the *using()* method on `QuerySet` objects. Individual objects can be saved to a specific database by providing a `using` argument when you call `save()`.

Model validation Model instances now have support for *validating their own data*, and both model and form fields now accept configurable lists of *validators* specifying reusable, encapsulated validation behavior. Note, however, that validation must still be performed explicitly. Simply invoking a model instance's `save()` method will not perform any validation of the instance's data.

Improved CSRF protection Django now has much improved protection against *Cross-Site Request Forgery (CSRF) attacks*. This type of attack occurs when a malicious Web site contains a link, a form button or some JavaScript that is intended to perform some action on your Web site, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, "login CSRF," where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered.

Messages framework Django now includes a robust and configurable *messages framework* with built-in support for cookie- and session-based messaging, for both anonymous and authenticated clients. The messages framework replaces the deprecated user message API and allows you to temporarily store messages in one request and retrieve them for display in a subsequent request (usually the next one).

Object-level permissions A foundation for specifying permissions at the per-object level has been added. Although there is no implementation of this in core, a custom authentication backend can provide this implementation and it will be used by `django.contrib.auth.models.User`. See the *authentication docs* for more information.

Permissions for anonymous users If you provide a custom auth backend with `supports_anonymous_user` set to `True`, AnonymousUser will check the backend for permissions, just like User already did. This is useful for centralizing permission handling - apps can always delegate the question of whether something is allowed or not to the authorization/authentication backend. See the *authentication docs* for more details.

Relaxed requirements for usernames The built-in `User` model's `username` field now allows a wider range of characters, including `@`, `+`, `.` and `-` characters.

Email backends You can now *configure the way that Django sends email*. Instead of using SMTP to send all email, you can now choose a configurable email backend to send messages. If your hosting provider uses a sandbox or some other non-SMTP technique for sending mail, you can now construct an email backend that will allow Django's standard *mail sending methods* to use those facilities.

This also makes it easier to debug mail sending. Django ships with backend implementations that allow you to send email to a *file*, to the *console*, or to *memory*. You can even configure all email to be *thrown away*.

"Smart" `if` tag The `if` tag has been upgraded to be much more powerful. First, we've added support for comparison operators. No longer will you have to type:

```
{% ifnotequal a b %}
  ...
{% endifnotequal %}
```

You can now do this:

```
{% if a != b %}
  ...
{% endif %}
```

There's really no reason to use `{% ifequal %}` or `{% ifnotequal %}` anymore, unless you're the nostalgic type.

The operators supported are `==`, `!=`, `<`, `>`, `<=`, `>=`, `in` and `not in`, all of which work like the Python operators, in addition to `and`, `or` and `not`, which were already supported.

Also, filters may now be used in the `if` expression. For example:

```
<div
  {% if user.email|lower == message.recipient|lower %}
    class="highlight"
  {% endif %}
>{{ message }}</div>
```

Template caching In previous versions of Django, every time you rendered a template, it would be reloaded from disk. In Django 1.2, you can use a *cached template loader* to load templates once, then cache the result for every subsequent render. This can lead to a significant performance improvement if your templates are broken into lots of smaller subtemplates (using the `{% extends %}` or `{% include %}` tags).

As a side effect, it is now much easier to support non-Django template languages. For more details, see the *notes on supporting non-Django template languages*.

Class-based template loaders As part of the changes made to introduce Template caching and following a general trend in Django, the template loaders API has been modified to use template loading mechanisms that are encapsulated in Python classes as opposed to functions, the only method available until Django 1.1.

All the template loaders *shipped with Django* have been ported to the new API but they still implement the function-based API and the template core machinery still accepts function-based loaders (builtin or third party) so there is no immediate need to modify your `TEMPLATE_LOADERS` setting in existing projects, things will keep working if you leave it untouched up to and including the Django 1.3 release.

If you have developed your own custom template loaders we suggest to consider porting them to a class-based implementation because the code for backwards compatibility with function-based loaders starts its deprecation process in Django 1.2 and will be removed in Django 1.4. There is a description of the API these loader classes must implement *here* and you can also examine the source code of the loaders shipped with Django.

Natural keys in fixtures Fixtures can now refer to remote objects using *Natural keys*. This lookup scheme is an alternative to the normal primary-key based object references in a fixture, improving readability and resolving problems referring to objects whose primary key value may not be predictable or known.

Fast failure for tests Both the `test` subcommand of `django-admin.py` and the `runtests.py` script used to run Django's own test suite now support a `--failfast` option. When specified, this option causes the test runner to exit after encountering a failure instead of continuing with the test run. In addition, the handling of `Ctrl-C` during a test run has been improved to trigger a graceful exit from the test run that reports details of the tests that were run before the interruption.

`BigIntegerField` Models can now use a 64-bit `BigIntegerField` type.

Improved localization Django's *internationalization framework* has been expanded with locale-aware formatting and form processing. That means, if enabled, dates and numbers on templates will be displayed using the format specified for the current locale. Django will also use localized formats when parsing data in forms. See *Format localization* for more details.

`readonly_fields` in `ModelAdmin` `django.contrib.admin.ModelAdmin.readonly_fields` has been added to enable non-editable fields in add/change pages for models and inlines. Field and calculated values can be displayed alongside editable fields.

Customizable syntax highlighting You can now use a `DJANGO_COLORS` environment variable to modify or disable the colors used by `django-admin.py` to provide *syntax highlighting*.

Syndication feeds as views *Syndication feeds* can now be used directly as views in your *URLconf*. This means that you can maintain complete control over the URL structure of your feeds. Like any other view, feeds views are passed a `request` object, so you can do anything you would normally do with a view, like user based access control, or making a feed a named URL.

GeoDjango The most significant new feature for *GeoDjango* in 1.2 is support for multiple spatial databases. As a result, the following *spatial database backends* are now included:

- `django.contrib.gis.db.backends.postgis`
- `django.contrib.gis.db.backends.mysql`
- `django.contrib.gis.db.backends.oracle`
- `django.contrib.gis.db.backends.spatialite`

GeoDjango now supports the rich capabilities added in the PostGIS 1.5 release. New features include suppport for the *geography type* and enabling of *distance queries* with non-point geometries on geographic coordinate systems.

Support for 3D geometry fields was added, and may be enabled by setting the `dim` keyword to 3 in your `GeometryField`. The `Extent3D` aggregate and `extent3d() GeoQuerySet` method were added as a part of this feature.

The following `GeoQuerySet` methods are new in 1.2:

- `force_rhr()`
- `reverse_geom()`
- `geohash()`

The *GEOS interface* was updated to use thread-safe C library functions when available on the platform.

The *GDAL interface* now allows the user to set a `spatial_filter` on the features returned when iterating over a `Layer`.

Finally, *GeoDjango's documentation* is now included with Django's and is no longer hosted separately at geodjango.org.

JavaScript-assisted handling of inline related objects in the admin If a user has JavaScript enabled in their browser, the interface for inline objects in the admin now allows inline objects to be dynamically added and removed. Users without JavaScript-enabled browsers will see no change in the behavior of inline objects.

New `now` template tag format specifier characters: `c` and `u` The argument to the `now` has gained two new format characters: `c` to specify that a datetime value should be formatted in ISO 8601 format, and `u` that allows output of the microseconds part of a datetime or time value.

These are also available in others parts like the `date` and `time` template filters, the `humanize` template tag library and the new format localization framework.

Backwards-incompatible changes in 1.2

Wherever possible the new features above have been introduced in a backwards-compatible manner per *our API stability policy* policy. This means that practically all existing code which worked with Django 1.1 will continue to work with Django 1.2; such code will, however, begin issuing warnings (see below for details).

However, a handful of features *have* changed in ways that, for some users, will be immediately backwards-incompatible. Those changes are detailed below.

CSRF Protection We've made large changes to the way CSRF protection works, detailed in *the CSRF documentation*. Here are the major changes you should be aware of:

- `CsrfResponseMiddleware` and `CsrfMiddleware` have been deprecated and will be removed completely in Django 1.4, in favor of a template tag that should be inserted into forms.

- All contrib apps use a `csrf_protect` decorator to protect the view. This requires the use of the `csrf_token` template tag in the template. If you have used custom templates for contrib views, you MUST READ THE UPGRADE INSTRUCTIONS to fix those templates.

Documentation removed

The upgrade notes have been removed in current Django docs. Please refer to the docs for Django 1.3 or older to find these instructions.

- `CsrfViewMiddleware` is included in `MIDDLEWARE_CLASSES` by default. This turns on CSRF protection by default, so views that accept POST requests need to be written to work with the middleware. Instructions on how to do this are found in the CSRF docs.
- All of the CSRF has moved from contrib to core (with backwards compatible imports in the old locations, which are deprecated and will cease to be supported in Django 1.4).

`get_db_prep_*()` methods on `Field` Prior to Django 1.2, a custom `Field` had the option of defining several functions to support conversion of Python values into database-compatible values. A custom field might look something like:

```
class CustomModelField(models.Field):
    # ...
    def db_type(self):
        # ...

    def get_db_prep_save(self, value):
        # ...

    def get_db_prep_value(self, value):
        # ...

    def get_db_prep_lookup(self, lookup_type, value):
        # ...
```

In 1.2, these three methods have undergone a change in prototype, and two extra methods have been introduced:

```
class CustomModelField(models.Field):
    # ...

    def db_type(self, connection):
        # ...

    def get_prep_value(self, value):
        # ...

    def get_prep_lookup(self, lookup_type, value):
        # ...

    def get_db_prep_save(self, value, connection):
        # ...

    def get_db_prep_value(self, value, connection, prepared=False):
        # ...

    def get_db_prep_lookup(self, lookup_type, value, connection, prepared=False):
        # ...
```

These changes are required to support multiple databases – `db_type` and `get_db_prep_*` can no longer make any assumptions regarding the database for which it is preparing. The `connection` argument now provides the preparation methods with the specific connection for which the value is being prepared.

The two new methods exist to differentiate general data-preparation requirements from requirements that are database-specific. The `prepared` argument is used to indicate to the database-preparation methods whether generic value preparation has been performed. If an unprepared (i.e., `prepared=False`) value is provided to the `get_db_prep_*()` calls, they should invoke the corresponding `get_prep_*()` calls to perform generic data preparation.

We've provided conversion functions that will transparently convert functions adhering to the old prototype into functions compatible with the new prototype. However, these conversion functions will be removed in Django 1.4, so you should upgrade your `Field` definitions to use the new prototype as soon as possible.

If your `get_db_prep_*()` methods made no use of the database connection, you should be able to upgrade by renaming `get_db_prep_value()` to `get_prep_value()` and `get_db_prep_lookup()` to `get_prep_lookup()`. If you require database specific conversions, then you will need to provide an implementation `get_db_prep_*` that uses the `connection` argument to resolve database-specific values.

Stateful template tags Template tags that store rendering state on their `Node` subclass have always been vulnerable to thread-safety and other issues; as of Django 1.2, however, they may also cause problems when used with the new *cached template loader*.

All of the built-in Django template tags are safe to use with the cached loader, but if you're using custom template tags that come from third party packages, or from your own code, you should ensure that the `Node` implementation for each tag is thread-safe. For more information, see *template tag thread safety considerations*.

You may also need to update your templates if you were relying on the implementation of Django's template tags *not* being thread safe. The `cycle` tag is the most likely to be affected in this way, especially when used in conjunction with the `include` tag. Consider the following template fragment:

```
{% for object in object_list %}
    {% include "subtemplate.html" %}
{% endfor %}
```

with a `subtemplate.html` that reads:

```
{% cycle 'even' 'odd' %}
```

Using the non-thread-safe, pre-Django 1.2 renderer, this would output:

```
even odd even odd ...
```

Using the thread-safe Django 1.2 renderer, you will instead get:

```
even even even even ...
```

This is because each rendering of the `include` tag is an independent rendering. When the `cycle` tag was not thread safe, the state of the `cycle` tag would leak between multiple renderings of the same `include`. Now that the `cycle` tag is thread safe, this leakage no longer occurs.

`user_passes_test`, `login_required` and `permission_required` `django.contrib.auth.decorators` provides the decorators `login_required`, `permission_required` and `user_passes_test`. Previously it was possible to use these decorators both on functions (where the first argument is 'request') and on methods (where the first argument is 'self', and the second argument is 'request'). Unfortunately, flaws were discovered in the code supporting this: it only works in limited circumstances, and produces errors that are very difficult to debug when it does not work.

For this reason, the 'auto adapt' behavior has been removed, and if you are using these decorators on methods, you will need to manually apply `django.utils.decorators.method_decorator()` to convert the decorator to one that works with methods. For example, you would change code from this:

```
class MyClass(object):

    @login_required
    def my_view(self, request):
        pass
```

to this:

```
from django.utils.decorators import method_decorator

class MyClass(object):

    @method_decorator(login_required)
    def my_view(self, request):
        pass
```

or:

```
from django.utils.decorators import method_decorator

login_required_m = method_decorator(login_required)

class MyClass(object):

    @login_required_m
    def my_view(self, request):
        pass
```

For those of you who've been following the development trunk, this change also applies to other decorators introduced since 1.1, including `csrf_protect`, `cache_control` and anything created using `decorator_from_middleware`.

`if` tag changes Due to new features in the `if` template tag, it no longer accepts 'and', 'or' and 'not' as valid **variable** names. Previously, these strings could be used as variable names. Now, the keyword status is always enforced, and template code such as `{% if not %}` or `{% if and %}` will throw a `TemplateSyntaxError`. Also, `in` is a new keyword and so is not a valid variable name in this tag.

`LazyObject` `LazyObject` is an undocumented-but-often-used utility class used for lazily wrapping other objects of unknown type.

In Django 1.1 and earlier, it handled introspection in a non-standard way, depending on wrapped objects implementing a public method named `get_all_members()`. Since this could easily lead to name clashes, it has been changed to use the standard Python introspection method, involving `__members__` and `__dir__()`.

If you used `LazyObject` in your own code and implemented the `get_all_members()` method for wrapped objects, you'll need to make a couple of changes:

First, if your class does not have special requirements for introspection (i.e., you have not implemented `__getattr__()` or other methods that allow for attributes not discoverable by normal mechanisms), you can simply remove the `get_all_members()` method. The default implementation on `LazyObject` will do the right thing.

If you have more complex requirements for introspection, first rename the `get_all_members()` method to `__dir__()`. This is the standard introspection method for Python 2.6 and above. If you require support for Python versions earlier than 2.6, add the following code to the class:

```
__members__ = property(lambda self: self.__dir__())
```

`__dict__` on model instances Historically, the `__dict__` attribute of a model instance has only contained attributes corresponding to the fields on a model.

In order to support multiple database configurations, Django 1.2 has added a `_state` attribute to object instances. This attribute will appear in `__dict__` for a model instance. If your code relies on iterating over `__dict__` to obtain a list of fields, you must now be prepared to handle or filter out the `_state` attribute.

Test runner exit status code The exit status code of the test runners (`tests/runtests.py` and `python manage.py test`) no longer represents the number of failed tests, because a failure of 256 or more tests resulted in a wrong exit status code. The exit status code for the test runner is now 0 for success (no failing tests) and 1 for any number of test failures. If needed, the number of test failures can be found at the end of the test runner's output.

Cookie encoding To fix bugs with cookies in Internet Explorer, Safari, and possibly other browsers, our encoding of cookie values was changed so that the comma and semicolon are treated as non-safe characters, and are therefore encoded as `\054` and `\073` respectively. This could produce backwards incompatibilities, especially if you are storing comma or semi-colon in cookies and have javascript code that parses and manipulates cookie values client-side.

`ModelForm.is_valid()` and `ModelForm.errors` Much of the validation work for ModelForms has been moved down to the model level. As a result, the first time you call `ModelForm.is_valid()`, access `ModelForm.errors` or otherwise trigger form validation, your model will be cleaned in-place. This conversion used to happen when the model was saved. If you need an unmodified instance of your model, you should pass a copy to the `ModelForm` constructor.

`BooleanField` on MySQL In previous versions of Django, a model's `BooleanField` under MySQL would return its value as either `1` or `0`, instead of `True` or `False`; for most people this wasn't a problem because `bool` is a subclass of `int` in Python. In Django 1.2, however, `BooleanField` on MySQL correctly returns a real `bool`. The only time this should ever be an issue is if you were expecting the `repr` of a `BooleanField` to print `1` or `0`.

Changes to the interpretation of `max_num` in FormSets As part of enhancements made to the handling of FormSets, the default value and interpretation of the `max_num` parameter to the *django.forms.formsets.formset_factory()* and *django.forms.models.modelformset_factory()* functions has changed slightly. This change also affects the way the `max_num` argument is *used for inline admin objects*

Previously, the default value for `max_num` was `0` (zero). FormSets then used the boolean value of `max_num` to determine if a limit was to be imposed on the number of generated forms. The default value of `0` meant that there was no default limit on the number of forms in a FormSet.

Starting with 1.2, the default value for `max_num` has been changed to `None`, and FormSets will differentiate between a value of `None` and a value of `0`. A value of `None` indicates that no limit on the number of forms is to be imposed; a value of `0` indicates that a maximum of 0 forms should be imposed. This doesn't necessarily mean that no forms will be displayed – see the *ModelFormSet documentation* for more details.

If you were manually specifying a value of `0` for `max_num`, you will need to update your FormSet and/or admin definitions.

See Also:

JavaScript-assisted handling of inline related objects in the admin

email_re An undocumented regular expression for validating email addresses has been moved from `django.form.fields` to `django.core.validators`. You will need to update your imports if you are using it.

Features deprecated in 1.2

Finally, Django 1.2 deprecates some features from earlier releases. These features are still supported, but will be gradually phased out over the next few release cycles.

Code taking advantage of any of the features below will raise a `PendingDeprecationWarning` in Django 1.2. This warning will be silent by default, but may be turned on using Python's `warnings` module, or by running Python with a `-Wd` or *-Wall* flag.

In Django 1.3, these warnings will become a `DeprecationWarning`, which is *not* silent. In Django 1.4 support for these features will be removed entirely.

See Also:

For more details, see the documentation *Django's release process* and our *deprecation timeline*.'

Specifying databases Prior to Django 1.2, Django used a number of settings to control access to a single database. Django 1.2 introduces support for multiple databases, and as a result the way you define database settings has changed.

Any existing Django settings file will continue to work as expected until Django 1.4. Until then, old-style database settings will be automatically translated to the new-style format.

In the old-style (pre 1.2) format, you had a number of `DATABASE_` settings in your settings file. For example:

```
DATABASE_NAME = 'test_db'
DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_USER = 'myusername'
DATABASE_PASSWORD = 's3krit'
```

These settings are now in a dictionary named `DATABASES`. Each item in the dictionary corresponds to a single database connection, with the name `'default'` describing the default database connection. The setting names have also been shortened. The previous sample settings would now look like this:

```
DATABASES = {
    'default': {
        'NAME': 'test_db',
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'USER': 'myusername',
        'PASSWORD': 's3krit',
    }
}
```

This affects the following settings:

Old setting	New Setting
DATABASE_ENGINE	ENGINE
DATABASE_HOST	HOST
DATABASE_NAME	NAME
DATABASE_OPTIONS	OPTIONS
DATABASE_PASSWORD	PASSWORD
DATABASE_PORT	PORT
DATABASE_USER	USER
TEST_DATABASE_CHARSET	TEST_CHARSET
TEST_DATABASE_COLLATION	TEST_COLLATION
TEST_DATABASE_NAME	TEST_NAME

These changes are also required if you have manually created a database connection using `DatabaseWrapper()` from your database backend of choice.

In addition to the change in structure, Django 1.2 removes the special handling for the built-in database backends. All database backends must now be specified by a fully qualified module name (i.e., `django.db.backends.postgresql_psycopg2`, rather than just `postgresql_psycopg2`).

`postgresql` database backend The `psycopg1` library has not been updated since October 2005. As a result, the `postgresql` database backend, which uses this library, has been deprecated.

If you are currently using the `postgresql` backend, you should migrate to using the `postgresql_psycopg2` backend. To update your code, install the `psycopg2` library and change the `DATABASE_ENGINE` setting to use `django.db.backends.postgresql_psycopg2`.

CSRF response-rewriting middleware `CsrfResponseMiddleware`, the middleware that automatically inserted CSRF tokens into `POST` forms in outgoing pages, has been deprecated in favor of a template tag method (see above), and will be removed completely in Django 1.4. `CsrfMiddleware`, which includes the functionality of `CsrfResponseMiddleware` and `CsrfViewMiddleware`, has likewise been deprecated.

Also, the CSRF module has moved from contrib to core, and the old imports are deprecated, as described in the upgrading notes.

Documentation removed

The upgrade notes have been removed in current Django docs. Please refer to the docs for Django 1.3 or older to find these instructions.

`SMTPConnection` The `SMTPConnection` class has been deprecated in favor of a generic email backend API. Old code that explicitly instantiated an instance of an SMTPConnection:

```
from django.core.mail import SMTPConnection
connection = SMTPConnection()
messages = get_notification_email()
connection.send_messages(messages)
```

...should now call `get_connection()` to instantiate a generic email connection:

```
from django.core.mail import get_connection
connection = get_connection()
messages = get_notification_email()
connection.send_messages(messages)
```

Depending on the value of the EMAIL_BACKEND setting, this may not return an SMTP connection. If you explicitly require an SMTP connection with which to send email, you can explicitly request an SMTP connection:

```
from django.core.mail import get_connection
connection = get_connection('django.core.mail.backends.smtp.EmailBackend')
messages = get_notification_email()
connection.send_messages(messages)
```

If your call to construct an instance of SMTPConnection required additional arguments, those arguments can be passed to the get_connection() call:

```
connection = get_connection('django.core.mail.backends.smtp.EmailBackend', hostname='localhost', port
```

User Messages API The API for storing messages in the user Message model (via user.message_set.create) is now deprecated and will be removed in Django 1.4 according to the standard *release process*.

To upgrade your code, you need to replace any instances of this:

```
user.message_set.create('a message')
```

...with the following:

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'a message')
```

Additionally, if you make use of the method, you need to replace the following:

```
for message in user.get_and_delete_messages():
    ...
```

...with:

```
from django.contrib import messages
for message in messages.get_messages(request):
    ...
```

For more information, see the full *messages documentation*. You should begin to update your code to use the new API immediately.

Date format helper functions django.utils.translation.get_date_formats() and django.utils.translation.get_partial_date_formats() have been deprecated in favor of the appropriate calls to django.utils.formats.get_format(), which is locale-aware when USE_L10N is set to True, and falls back to default settings if set to False.

To get the different date formats, instead of writing this:

```
from django.utils.translation import get_date_formats
date_format, datetime_format, time_format = get_date_formats()
```

...use:

```
from django.utils import formats
date_format = formats.get_format('DATE_FORMAT')
datetime_format = formats.get_format('DATETIME_FORMAT')
time_format = formats.get_format('TIME_FORMAT')
```

Or, when directly formatting a date value:

```
from django.utils import formats
value_formatted = formats.date_format(value, 'DATETIME_FORMAT')
```

The same applies to the globals found in `django.forms.fields`:

- `DEFAULT_DATE_INPUT_FORMATS`
- `DEFAULT_TIME_INPUT_FORMATS`
- `DEFAULT_DATETIME_INPUT_FORMATS`

Use `django.utils.formats.get_format()` to get the appropriate formats.

Function-based test runners Django 1.2 changes the test runner tools to use a class-based approach. Old style function-based test runners will still work, but should be updated to use the new *class-based runners*.

`Feed` in `django.contrib.syndication.feeds` The `django.contrib.syndication.feeds.Feed` class has been replaced by the `django.contrib.syndication.views.Feed` class. The old `feeds.Feed` class is deprecated, and will be removed in Django 1.4.

The new class has an almost identical API, but allows instances to be used as views. For example, consider the use of the old framework in the following *URLconf*:

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

Using the new Feed class, these feeds can be deployed directly as views:

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

urlpatterns = patterns('',
    # ...
    (r'^feeds/latest/$', LatestEntries()),
    (r'^feeds/categories/(?P<category_id>\d+)/$', LatestEntriesByCategory()),
    # ...
)
```

If you currently use the `feed()` view, the `LatestEntries` class would often not need to be modified apart from subclassing the new `Feed` class. The exception is if Django was automatically working out the name of the template to use to render the feed's description and title elements (if you were not specifying the `title_template` and `description_template` attributes). You should ensure that you always specify `title_template` and `description_template` attributes, or provide `item_title()` and `item_description()` methods.

However, `LatestEntriesByCategory` uses the `get_object()` method with the `bits` argument to specify a specific category to show. In the new `Feed` class, `get_object()` method takes a `request` and arguments from the URL, so it would look like this:

```
from django.contrib.syndication.views import Feed
from django.shortcuts import get_object_or_404
from myproject.models import Category

class LatestEntriesByCategory(Feed):
    def get_object(self, request, category_id):
        return get_object_or_404(Category, id=category_id)

    # ...
```

Additionally, the `get_feed()` method on `Feed` classes now take different arguments, which may impact you if you use the `Feed` classes directly. Instead of just taking an optional `url` argument, it now takes two arguments: the object returned by its own `get_object()` method, and the current `request` object.

To take into account `Feed` classes not being initialized for each request, the `__init__()` method now takes no arguments by default. Previously it would have taken the `slug` from the URL and the `request` object.

In accordance with RSS best practices, RSS feeds will now include an `atom:link` element. You may need to update your tests to take this into account.

For more information, see the full *syndication framework documentation*.

Technical message IDs Up to version 1.1 Django used technical message IDs to provide localizers the possibility to translate date and time formats. They were translatable *translation strings* that could be recognized because they were all upper case (for example `DATETIME_FORMAT`, `DATE_FORMAT`, `TIME_FORMAT`). They have been deprecated in favor of the new *Format localization* infrastructure that allows localizers to specify that information in a `formats.py` file in the corresponding `django/conf/locale/<locale name>/` directory.

GeoDjango To allow support for multiple databases, the GeoDjango database internals were changed substantially. The largest backwards-incompatible change is that the module `django.contrib.gis.db.backend` was renamed to `django.contrib.gis.db.backends`, where the full-fledged *spatial database backends* now exist. The following sections provide information on the most-popular APIs that were affected by these changes.

`SpatialBackend` Prior to the creation of the separate spatial backends, the `django.contrib.gis.db.backend.SpatialBackend` object was provided as an abstraction to introspect on the capabilities of the spatial database. All of the attributes and routines provided by `SpatialBackend` are now a part of the `ops` attribute of the database backend.

The old module `django.contrib.gis.db.backend` is still provided for backwards-compatibility access to a `SpatialBackend` object, which is just an alias to the `ops` module of the *default* spatial database connection.

Users that were relying on undocumented modules and objects within `django.contrib.gis.db.backend`, rather the abstractions provided by `SpatialBackend`, are required to modify their code. For example, the following import which would work in 1.1 and below:

```
from django.contrib.gis.db.backend.postgis import PostGISAdaptor
```

Would need to be changed:

```
from django.db import connection
PostGISAdaptor = connection.ops.Adapter
```

`SpatialRefSys` and `GeometryColumns` models In previous versions of GeoDjango, `django.contrib.gis.db.models` had `SpatialRefSys` and `GeometryColumns` models for querying the OGC spatial metadata tables `spatial_ref_sys` and `geometry_columns`, respectively.

While these aliases are still provided, they are only for the *default* database connection and exist only if the default connection is using a supported spatial database backend.

Note: Because the table structure of the OGC spatial metadata tables differs across spatial databases, the `SpatialRefSys` and `GeometryColumns` models can no longer be associated with the `gis` application name. Thus, no models will be returned when using the `get_models` method in the following example:

```
>>> from django.db.models import get_app, get_models
>>> get_models(get_app('gis'))
[]
```

To get the correct `SpatialRefSys` and `GeometryColumns` for your spatial database use the methods provided by the spatial backend:

```
>>> from django.db import connections
>>> SpatialRefSys = connections['my_spatialite'].ops.spatial_ref_sys()
>>> GeometryColumns = connections['my_postgis'].ops.geometry_columns()
```

Note: When using the models returned from the `spatial_ref_sys()` and `geometry_columns()` method, you'll still need to use the correct database alias when querying on the non-default connection. In other words, to ensure that the models in the example above use the correct database:

```
sr_qs = SpatialRefSys.objects.using('my_spatialite').filter(...)
gc_qs = GeometryColumns.objects.using('my_postgis').filter(...)
```

Language code `no` The currently used language code for Norwegian Bokmål `no` is being replaced by the more common language code `nb`.

Function-based template loaders Django 1.2 changes the template loading mechanism to use a class-based approach. Old style function-based template loaders will still work, but should be updated to use the new *class-based template loaders*.

9.1.5 1.1 release

Django 1.1.4 release notes

Welcome to Django 1.1.4!

This is the fourth "bugfix" release in the Django 1.1 series, improving the stability and performance of the Django 1.1 codebase.

With one exception, Django 1.1.4 maintains backwards compatibility with Django 1.1.3. It also contains a number of fixes and other improvements. Django 1.1.4 is a recommended upgrade for any development or deployment currently using or targeting Django 1.1.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.1 branch, see the *Django 1.1 release notes*.

Backwards incompatible changes

CSRF exception for AJAX requests Django includes a CSRF-protection mechanism, which makes use of a token inserted into outgoing forms. Middleware then checks for the token's presence on form submission, and validates it.

Prior to Django 1.2.5, our CSRF protection made an exception for AJAX requests, on the following basis:

- Many AJAX toolkits add an X-Requested-With header when using XMLHttpRequest.
- Browsers have strict same-origin policies regarding XMLHttpRequest.
- In the context of a browser, the only way that a custom header of this nature can be added is with XMLHttpRequest.

Therefore, for ease of use, we did not apply CSRF checks to requests that appeared to be AJAX on the basis of the X-Requested-With header. The Ruby on Rails web framework had a similar exemption.

Recently, engineers at Google made members of the Ruby on Rails development team aware of a combination of browser plugins and redirects which can allow an attacker to provide custom HTTP headers on a request to any website. This can allow a forged request to appear to be an AJAX request, thereby defeating CSRF protection which trusts the same-origin nature of AJAX requests.

Michael Koziarski of the Rails team brought this to our attention, and we were able to produce a proof-of-concept demonstrating the same vulnerability in Django's CSRF handling.

To remedy this, Django will now apply full CSRF validation to all requests, regardless of apparent AJAX origin. This is technically backwards-incompatible, but the security risks have been judged to outweigh the compatibility concerns in this case.

Additionally, Django will now accept the CSRF token in the custom HTTP header X-CSRFTOKEN, as well as in the form submission itself, for ease of use with popular JavaScript toolkits which allow insertion of custom headers into all AJAX requests.

Please see the *CSRF docs for example jQuery code* that demonstrates this technique, ensuring that you are looking at the documentation for your version of Django, as the exact code necessary is different for some older versions of Django.

Django 1.1.3 release notes

Welcome to Django 1.1.3!

This is the third "bugfix" release in the Django 1.1 series, improving the stability and performance of the Django 1.1 codebase.

With one exception, Django 1.1.3 maintains backwards compatibility with Django 1.1.2. It also contains a number of fixes and other improvements. Django 1.1.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.1.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.1 branch, see the *Django 1.1 release notes*.

Backwards incompatible changes

Restricted filters in admin interface The Django administrative interface, django.contrib.admin, supports filtering of displayed lists of objects by fields on the corresponding models, including across database-level relationships. This is implemented by passing lookup arguments in the querystring portion of the URL, and options on the ModelAdmin class allow developers to specify particular fields or relationships which will generate automatic links for filtering.

One historically-undocumented and -unofficially-supported feature has been the ability for a user with sufficient knowledge of a model's structure and the format of these lookup arguments to invent useful new filters on the fly by manipulating the querystring.

However, it has been demonstrated that this can be abused to gain access to information outside of an admin user's permissions; for example, an attacker with access to the admin and sufficient knowledge of model structure and relations could construct query strings which – with repeated use of regular-expression lookups supported by the Django database API – expose sensitive information such as users' password hashes.

To remedy this, django.contrib.admin will now validate that querystring lookup arguments either specify only fields on the model being viewed, or cross relations which have been explicitly whitelisted by the application developer using the pre-existing mechanism mentioned above. This is backwards-incompatible for any users relying on the prior ability to insert arbitrary lookups.

Django 1.1.2 release notes

Welcome to Django 1.1.2!

This is the second "bugfix" release in the Django 1.1 series, improving the stability and performance of the Django 1.1 codebase.

Django 1.1.2 maintains backwards compatibility with Django 1.1.0, but contain a number of fixes and other improvements. Django 1.1.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.1.

For full details on the new features, backwards incompatibilities, and deprecated features in the 1.1 branch, see the *Django 1.1 release notes*.

Backwards-incompatible changes in 1.1.2

Test runner exit status code The exit status code of the test runners (`tests/runtests.py` and `python manage.py test`) no longer represents the number of failed tests, since a failure of 256 or more tests resulted in a wrong exit status code. The exit status code for the test runner is now 0 for success (no failing tests) and 1 for any number of test failures. If needed, the number of test failures can be found at the end of the test runner's output.

Cookie encoding To fix bugs with cookies in Internet Explorer, Safari, and possibly other browsers, our encoding of cookie values was changed so that the characters comma and semi-colon are treated as non-safe characters, and are therefore encoded as `\054` and `\073` respectively. This could produce backwards incompatibilities, especially if you are storing comma or semi-colon in cookies and have javascript code that parses and manipulates cookie values client-side.

One new feature

Ordinarily, a point release would not include new features, but in the case of Django 1.1.2, we have made an exception to this rule. Django 1.2 (the next major release of Django) will contain a feature that will improve protection against Cross-Site Request Forgery (CSRF) attacks. This feature requires the use of a new `csrf_token` template tag in all forms that Django renders.

To make it easier to support both 1.1.X and 1.2.X versions of Django with the same templates, we have decided to introduce the `csrf_token` template tag to the 1.1.X branch. In the 1.1.X branch, `csrf_token` does nothing - it has no effect on templates or form processing. However, it means that the same template will work with Django 1.2.

Django 1.1 release notes

July 29, 2009

Welcome to Django 1.1!

Django 1.1 includes a number of nifty new features, lots of bug fixes, and an easy upgrade path from Django 1.0.

Backwards-incompatible changes in 1.1

Django has a policy of *API stability*. This means that, in general, code you develop against Django 1.0 should continue to work against 1.1 unchanged. However, we do sometimes make backwards-incompatible changes if they're necessary to resolve bugs, and there are a handful of such (minor) changes between Django 1.0 and Django 1.1.

Before upgrading to Django 1.1 you should double-check that the following changes don't impact you, and upgrade your code if they do.

Changes to constraint names Django 1.1 modifies the method used to generate database constraint names so that names are consistent regardless of machine word size. This change is backwards incompatible for some users.

If you are using a 32-bit platform, you're off the hook; you'll observe no differences as a result of this change.

However, **users on 64-bit platforms may experience some problems** using the `reset` management command. Prior to this change, 64-bit platforms would generate a 64-bit, 16 character digest in the constraint name; for example:

```
ALTER TABLE myapp_sometable ADD CONSTRAINT object_id_refs_id_5e8f10c132091d1e FOREIGN KEY ...
```

Following this change, all platforms, regardless of word size, will generate a 32-bit, 8 character digest in the constraint name; for example:

```
ALTER TABLE myapp_sometable ADD CONSTRAINT object_id_refs_id_32091d1e FOREIGN KEY ...
```

As a result of this change, you will not be able to use the `reset` management command on any table made by a 64-bit machine. This is because the the new generated name will not match the historically generated name; as a result, the SQL constructed by the reset command will be invalid.

If you need to reset an application that was created with 64-bit constraints, you will need to manually drop the old constraint prior to invoking `reset`.

Test cases are now run in a transaction Django 1.1 runs tests inside a transaction, allowing better test performance (see test performance improvements for details).

This change is slightly backwards incompatible if existing tests need to test transactional behavior, if they rely on invalid assumptions about the test environment, or if they require a specific test case ordering.

For these cases, `TransactionTestCase` can be used instead. This is a just a quick fix to get around test case errors revealed by the new rollback approach; in the long-term tests should be rewritten to correct the test case.

Removed `SetRemoteAddrFromForwardedFor` middleware For convenience, Django 1.0 included an optional middleware class – `django.middleware.http.SetRemoteAddrFromForwardedFor` – which updated the value of `REMOTE_ADDR` based on the HTTP `X-Forwarded-For` header commonly set by some proxy configurations.

It has been demonstrated that this mechanism cannot be made reliable enough for general-purpose use, and that (despite documentation to the contrary) its inclusion in Django may lead application developers to assume that the value of `REMOTE_ADDR` is "safe" or in some way reliable as a source of authentication.

While not directly a security issue, we've decided to remove this middleware with the Django 1.1 release. It has been replaced with a class that does nothing other than raise a `DeprecationWarning`.

If you've been relying on this middleware, the easiest upgrade path is:

- Examine the code as it existed before it was removed.

- Verify that it works correctly with your upstream proxy, modifying it to support your particular proxy (if necessary).
- Introduce your modified version of `SetRemoteAddrFromForwardedFor` as a piece of middleware in your own project.

Names of uploaded files are available later In Django 1.0, files uploaded and stored in a model's `FileField` were saved to disk before the model was saved to the database. This meant that the actual file name assigned to the file was available before saving. For example, it was available in a model's pre-save signal handler.

In Django 1.1 the file is saved as part of saving the model in the database, so the actual file name used on disk cannot be relied on until *after* the model has been saved.

Changes to how model formsets are saved In Django 1.1, `BaseModelFormSet` now calls `ModelForm.save()`.

This is backwards-incompatible if you were modifying `self.initial` in a model formset's `__init__`, or if you relied on the internal `_total_form_count` or `_initial_form_count` attributes of BaseFormSet. Those attributes are now public methods.

Fixed the `join` filter's escaping behavior The `join` filter no longer escapes the literal value that is passed in for the connector.

This is backwards incompatible for the special situation of the literal string containing one of the five special HTML characters. Thus, if you were writing `{{ foo|join:"&" }}`, you now have to write `{{ foo|join:"&" }}`.

The previous behavior was a bug and contrary to what was documented and expected.

Permanent redirects and the `redirect_to()` generic view Django 1.1 adds a `permanent` argument to the `django.views.generic.simple.redirect_to()` view. This is technically backwards-incompatible if you were using the `redirect_to` view with a format-string key called 'permanent', which is highly unlikely.

Features deprecated in 1.1

One feature has been marked as deprecated in Django 1.1:

- You should no longer use `AdminSite.root()` to register that admin views. That is, if your URLconf contains the line:

  ```
  (r'^admin/(.*)', admin.site.root),
  ```

  You should change it to read:

  ```
  (r'^admin/', include(admin.site.urls)),
  ```

You should begin to remove use of this feature from your code immediately.

`AdminSite.root` will raise a `PendingDeprecationWarning` if used in Django 1.1. This warning is hidden by default. In Django 1.2, this warning will be upgraded to a `DeprecationWarning`, which will be displayed loudly. Django 1.3 will remove `AdminSite.root()` entirely.

For more details on our deprecation policies and strategy, see *Django's release process*.

What's new in Django 1.1

Quite a bit: since Django 1.0, we've made 1,290 code commits, fixed 1,206 bugs, and added roughly 10,000 lines of documentation.

The major new features in Django 1.1 are:

ORM improvements Two major enhancements have been added to Django's object-relational mapper (ORM): aggregate support, and query expressions.

Aggregate support It's now possible to run SQL aggregate queries (i.e. `COUNT()`, `MAX()`, `MIN()`, etc.) from within Django's ORM. You can choose to either return the results of the aggregate directly, or else annotate the objects in a `QuerySet` with the results of the aggregate query.

This feature is available as new `QuerySet.aggregate()`‘() and `QuerySet.annotate()`‘() methods, and is covered in detail in *the ORM aggregation documentation*.

Query expressions Queries can now refer to a another field on the query and can traverse relationships to refer to fields on related models. This is implemented in the new `F` object; for full details, including examples, consult the *documentation for F expressions*.

Model improvements A number of features have been added to Django's model layer:

"Unmanaged" models You can now control whether or not Django manages the life-cycle of the database tables for a model using the `managed` model option. This defaults to `True`, meaning that Django will create the appropriate database tables in `syncdb` and remove them as part of the `reset` command. That is, Django *manages* the database table's lifecycle.

If you set this to `False`, however, no database table creating or deletion will be automatically performed for this model. This is useful if the model represents an existing table or a database view that has been created by some other means.

For more details, see the documentation for the `managed` option.

Proxy models You can now create *proxy models*: subclasses of existing models that only add Python-level (rather than database-level) behavior and aren't represented by a new table. That is, the new model is a *proxy* for some underlying model, which stores all the real data.

All the details can be found in the *proxy models documentation*. This feature is similar on the surface to unmanaged models, so the documentation has an explanation of *how proxy models differ from unmanaged models*.

Deferred fields In some complex situations, your models might contain fields which could contain a lot of data (for example, large text fields), or require expensive processing to convert them to Python objects. If you know you don't need those particular fields, you can now tell Django not to retrieve them from the database.

You'll do this with the new queryset methods `defer()` and `only()`.

Testing improvements A few notable improvements have been made to the *testing framework*.

Test performance improvements Tests written using Django's *testing framework* now run dramatically faster (as much as 10 times faster in many cases).

This was accomplished through the introduction of transaction-based tests: when using `django.test.TestCase`, your tests will now be run in a transaction which is rolled back when finished, instead of by flushing and re-populating the database. This results in an immense speedup for most types of unit tests. See the documentation for `TestCase` and `TransactionTestCase` for a full description, and some important notes on database support.

Test client improvements A couple of small – but highly useful – improvements have been made to the test client:

- The test `Client` now can automatically follow redirects with the `follow` argument to `Client.get()` and `Client.post()`. This makes testing views that issue redirects simpler.
- It's now easier to get at the template context in the response returned the test client: you'll simply access the context as `request.context[key]`. The old way, which treats `request.context` as a list of contexts, one for each rendered template in the inheritance chain, is still available if you need it.

New admin features Django 1.1 adds a couple of nifty new features to Django's admin interface:

Editable fields on the change list You can now make fields editable on the admin list views via the new *list_editable* admin option. These fields will show up as form widgets on the list pages, and can be edited and saved in bulk.

Admin "actions" You can now define *admin actions* that can perform some action to a group of models in bulk. Users will be able to select objects on the change list page and then apply these bulk actions to all selected objects.

Django ships with one pre-defined admin action to delete a group of objects in one fell swoop.

Conditional view processing Django now has much better support for *conditional view processing* using the standard `ETag` and `Last-Modified` HTTP headers. This means you can now easily short-circuit view processing by testing less-expensive conditions. For many views this can lead to a serious improvement in speed and reduction in bandwidth.

URL namespaces Django 1.1 improves *named URL patterns* with the introduction of URL "namespaces."

In short, this feature allows the same group of URLs, from the same application, to be included in a Django URLConf multiple times, with varying (and potentially nested) named prefixes which will be used when performing reverse resolution. In other words, reusable applications like Django's admin interface may be registered multiple times without URL conflicts.

For full details, see *the documentation on defining URL namespaces*.

GeoDjango In Django 1.1, GeoDjango (i.e. `django.contrib.gis`) has several new features:

- Support for SpatiaLite – a spatial database for SQLite – as a spatial backend.
- Geographic aggregates (`Collect`, `Extent`, `MakeLine`, `Union`) and `F` expressions.
- New `GeoQuerySet` methods: `collect`, `geojson`, and `snap_to_grid`.
- A new list interface methods for `GEOSGeometry` objects.

For more details, see the GeoDjango documentation.

Other improvements Other new features and changes introduced since Django 1.0 include:

- The *CSRF protection middleware* has been split into two classes – `CsrfViewMiddleware` checks incoming requests, and `CsrfResponseMiddleware` processes outgoing responses. The combined `CsrfMiddleware` class (which does both) remains for backwards-compatibility, but using the split classes is now recommended in order to allow fine-grained control of when and where the CSRF processing takes place.
- `reverse()` and code which uses it (e.g., the `{% url %}` template tag) now works with URLs in Django's administrative site, provided that the admin URLs are set up via `include(admin.site.urls)` (sending admin requests to the `admin.site.root` view still works, but URLs in the admin will not be "reversible" when configured this way).
- The `include()` function in Django URLconf modules can now accept sequences of URL patterns (generated by `patterns()`) in addition to module names.
- Instances of Django forms (see *the forms overview*) now have two additional methods, `hidden_fields()` and `visible_fields()`, which return the list of hidden – i.e., `<input type="hidden">` – and visible fields on the form, respectively.
- The `redirect_to` generic view now accepts an additional keyword argument `permanent`. If `permanent` is `True`, the view will emit an HTTP permanent redirect (status code 301). If `False`, the view will emit an HTTP temporary redirect (status code 302).
- A new database lookup type – `week_day` – has been added for `DateField` and `DateTimeField`. This type of lookup accepts a number between 1 (Sunday) and 7 (Saturday), and returns objects where the field value matches that day of the week. See *the full list of lookup types* for details.
- The `{% for %}` tag in Django's template language now accepts an optional `{% empty %}` clause, to be displayed when `{% for %}` is asked to loop over an empty sequence. See *the list of built-in template tags* for examples of this.
- The `dumpdata` management command now accepts individual model names as arguments, allowing you to export the data just from particular models.
- There's a new `safeseq` template filter which works just like `safe` for lists, marking each item in the list as safe.
- *Cache backends* now support `incr()` and `decr()` commands to increment and decrement the value of a cache key. On cache backends that support atomic increment/decrement – most notably, the memcached backend – these operations will be atomic, and quite fast.
- Django now can *easily delegate authentication to the Web server* via a new authentication backend that supports the standard `REMOTE_USER` environment variable used for this purpose.
- There's a new `django.shortcuts.redirect()` function that makes it easier to issue redirects given an object, a view name, or a URL.
- The `postgresql_psycopg2` backend now supports *native PostgreSQL autocommit*. This is an advanced, PostgreSQL-specific feature, that can make certain read-heavy applications a good deal faster.

What's next?

We'll take a short break, and then work on Django 1.2 will begin – no rest for the weary! If you'd like to help, discussion of Django development, including progress toward the 1.2 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. Feel free to join the discussions!

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

And that's the way it is.

9.1.6 1.0 release

Django 1.0.2 release notes

Welcome to Django 1.0.2!

This is the second "bugfix" release in the Django 1.0 series, improving the stability and performance of the Django 1.0 codebase. As such, Django 1.0.2 contains no new features (and, pursuant to *our compatibility policy*, maintains backwards compatibility with Django 1.0.0), but does contain a number of fixes and other improvements. Django 1.0.2 is a recommended upgrade for any development or deployment currently using or targeting Django 1.0.

Fixes and improvements in Django 1.0.2

The primary reason behind this release is to remedy an issue in the recently-released Django 1.0.1; the packaging scripts used for Django 1.0.1 omitted some directories from the final release package, including one directory required by `django.contrib.gis` and part of Django's unit-test suite.

Django 1.0.2 contains updated packaging scripts, and the release package contains the directories omitted from Django 1.0.1. As such, this release contains all of the fixes and improvements from Django 1.0.1; see *the Django 1.0.1 release notes* for details.

Additionally, in the period since Django 1.0.1 was released:

- Updated Hebrew and Danish translations have been added.
- The default `__repr__` method of Django models has been made more robust in the face of bad Unicode data coming from the `__unicode__` method; rather than raise an exception in such cases, `repr()` will now contain the string "[Bad Unicode data]" in place of the invalid Unicode.
- A bug involving the interaction of Django's `SafeUnicode` class and the MySQL adapter has been resolved; `SafeUnicode` instances (generated, for example, by template rendering) can now be assigned to model attributes and saved to MySQL without requiring an explicit intermediate cast to `unicode`.
- A bug affecting filtering on a nullable `DateField` in SQLite has been resolved.
- Several updates and improvements have been made to Django's documentation.

Django 1.0.1 release notes

Welcome to Django 1.0.1!

This is the first "bugfix" release in the Django 1.0 series, improving the stability and performance of the Django 1.0 codebase. As such, Django 1.0.1 contains no new features (and, pursuant to *our compatibility policy*, maintains backwards compatibility with Django 1.0), but does contain a number of fixes and other improvements. Django 1.0.1 is a recommended upgrade for any development or deployment currently using or targeting Django 1.0.

Fixes and improvements in Django 1.0.1

Django 1.0.1 contains over two hundred fixes to the original Django 1.0 codebase; full details of every fix are available in the history of the 1.0.X branch, but here are some of the highlights:

- Several fixes in `django.contrib.comments`, pertaining to RSS feeds of comments, default ordering of comments and the XHTML and internationalization of the default templates for comments.
- Multiple fixes for Django's support of Oracle databases, including pagination support for GIS QuerySets, more efficient slicing of results and improved introspection of existing databases.
- Several fixes for query support in the Django object-relational mapper, including repeated setting and resetting of ordering and fixes for working with `INSERT`-only queries.
- Multiple fixes for inline forms in formsets.
- Multiple fixes for `unique` and `unique_together` model constraints in automatically-generated forms.
- Fixed support for custom callable `upload_to` declarations when handling file uploads through automatically-generated forms.
- Fixed support for sorting an admin change list based on a callable attributes in `list_display`.
- A fix to the application of autoescaping for literal strings passed to the `join` template filter. Previously, literal strings passed to `join` were automatically escaped, contrary to *the documented behavior for autoescaping and literal strings*. Literal strings passed to `join` are no longer automatically escaped, meaning you must now manually escape them; this is an incompatibility if you were relying on this bug, but not if you were relying on escaping behaving as documented.
- Improved and expanded translation files for many of the languages Django supports by default.
- And as always, a large number of improvements to Django's documentation, including both corrections to existing documents and expanded and new documentation.

Django 1.0 release notes

Welcome to Django 1.0!

We've been looking forward to this moment for over three years, and it's finally here. Django 1.0 represents a the largest milestone in Django's development to date: a Web framework that a group of perfectionists can truly be proud of.

Django 1.0 represents over three years of community development as an Open Source project. Django's received contributions from hundreds of developers, been translated into fifty languages, and today is used by developers on every continent and in every kind of job.

An interesting historical note: when Django was first released in July 2005, the initial released version of Django came from an internal repository at revision number 8825. Django 1.0 represents revision 8961 of our public repository. It seems fitting that our 1.0 release comes at the moment where community contributions overtake those made privately.

Stability and forwards-compatibility

The release of Django 1.0 comes with a promise of API stability and forwards-compatibility. In a nutshell, this means that code you develop against Django 1.0 will continue to work against 1.1 unchanged, and you should need to make only minor changes for any 1.X release.

See the *API stability guide* for full details.

Backwards-incompatible changes

Django 1.0 has a number of backwards-incompatible changes from Django 0.96. If you have apps written against Django 0.96 that you need to port, see our detailed porting guide:

Porting your apps from Django 0.96 to 1.0 Django 1.0 breaks compatibility with 0.96 in some areas.

This guide will help you port 0.96 projects and apps to 1.0. The first part of this document includes the common changes needed to run with 1.0. If after going through the first part your code still breaks, check the section Less-common Changes for a list of a bunch of less-common compatibility issues.

See Also:

The *1.0 release notes*. That document explains the new features in 1.0 more deeply; the porting guide is more concerned with helping you quickly update your code.

Common changes This section describes the changes between 0.96 and 1.0 that most users will need to make.

Use Unicode Change string literals (`'foo'`) into Unicode literals (`u'foo'`). Django now uses Unicode strings throughout. In most places, raw strings will continue to work, but updating to use Unicode literals will prevent some obscure problems.

See *Unicode data* for full details.

Models Common changes to your models file:

Rename `maxlength` to `max_length` Rename your `maxlength` argument to `max_length` (this was changed to be consistent with form fields):

Replace `__str__` with `__unicode__` Replace your model's `__str__` function with a `__unicode__` method, and make sure you use Unicode (`u'foo'`) in that method.

Remove `prepopulated_from` Remove the `prepopulated_from` argument on model fields. It's no longer valid and has been moved to the `ModelAdmin` class in `admin.py`. See the admin, below, for more details about changes to the admin.

Remove `core` Remove the `core` argument from your model fields. It is no longer necessary, since the equivalent functionality (part of *inline editing*) is handled differently by the admin interface now. You don't have to worry about inline editing until you get to the admin section, below. For now, remove all references to `core`.

Replace `class Admin:` with `admin.py` Remove all your inner `class Admin` declarations from your models. They won't break anything if you leave them, but they also won't do anything. To register apps with the admin you'll move those declarations to an `admin.py` file; see the admin below for more details.

See Also:

A contributor to djangosnippets has written a script that'll scan your models.py and generate a corresponding admin.py.

Example Below is an example `models.py` file with all the changes you'll need to make:

Old (0.96) `models.py`:

```
class Author(models.Model):
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=30)
    slug = models.CharField(maxlength=60, prepopulate_from=('first_name', 'last_name'))

    class Admin:
        list_display = ['first_name', 'last_name']

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

New (1.0) `models.py`:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    slug = models.CharField(max_length=60)

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

New (1.0) `admin.py`:

```
from django.contrib import admin
from models import Author

class AuthorAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name']
    prepopulated_fields = {
        'slug': ('first_name', 'last_name')
    }

admin.site.register(Author, AuthorAdmin)
```

The Admin One of the biggest changes in 1.0 is the new admin. The Django administrative interface (`django.contrib.admin`) has been completely refactored; admin definitions are now completely decoupled from model definitions, the framework has been rewritten to use Django's new form-handling library and redesigned with extensibility and customization in mind.

Practically, this means you'll need to rewrite all of your `class Admin` declarations. You've already seen in models above how to replace your `class Admin` with a `admin.site.register()` call in an `admin.py` file. Below are some more details on how to rewrite that `Admin` declaration into the new syntax.

Use new inline syntax The new `edit_inline` options have all been moved to `admin.py`. Here's an example:

Old (0.96):

```
class Parent(models.Model):
    ...

class Child(models.Model):
    parent = models.ForeignKey(Parent, edit_inline=models.STACKED, num_in_admin=3)
```

New (1.0):

```
class ChildInline(admin.StackedInline):
    model = Child
    extra = 3

class ParentAdmin(admin.ModelAdmin):
    model = Parent
    inlines = [ChildInline]

admin.site.register(Parent, ParentAdmin)
```

See *InlineModelAdmin objects* for more details.

Simplify `fields`, or use `fieldsets` The old `fields` syntax was quite confusing, and has been simplified. The old syntax still works, but you'll need to use `fieldsets` instead.

Old (0.96):

```
class ModelOne(models.Model):
    ...

    class Admin:
        fields = (
            (None, {'fields': ('foo','bar')}),
        )

class ModelTwo(models.Model):
    ...

    class Admin:
        fields = (
            ('group1', {'fields': ('foo','bar'),   'classes': 'collapse'}),
            ('group2', {'fields': ('spam','eggs'), 'classes': 'collapse wide'}),
        )
```

New (1.0):

```
class ModelOneAdmin(admin.ModelAdmin):
    fields = ('foo', 'bar')

class ModelTwoAdmin(admin.ModelAdmin):
    fieldsets = (
        ('group1', {'fields': ('foo','bar'),   'classes': 'collapse'}),
        ('group2', {'fields': ('spam','eggs'), 'classes': 'collapse wide'}),
    )
```

See Also:

- More detailed information about the changes and the reasons behind them can be found on the NewformsAdminBranch wiki page
- The new admin comes with a ton of new features; you can read about them in the *admin documentation*.

URLs

Update your root `urls.py` If you're using the admin site, you need to update your root `urls.py`.

Old (0.96) `urls.py`:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^admin/', include('django.contrib.admin.urls')),

    # ... the rest of your URLs here ...
)
```

New (1.0) `urls.py`:

```
from django.conf.urls.defaults import *

# The next two lines enable the admin and load each admin.py file:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),

    # ... the rest of your URLs here ...
)
```

Views

Use `django.forms` instead of `newforms` Replace `django.newforms` with `django.forms` – Django 1.0 renamed the `newforms` module (introduced in 0.96) to plain old `forms`. The `oldforms` module was also removed.

If you're already using the `newforms` library, and you used our recommended `import` statement syntax, all you have to do is change your import statements.

Old:

```
from django import newforms as forms
```

New:

```
from django import forms
```

If you're using the old forms system (formerly known as `django.forms` and `django.oldforms`), you'll have to rewrite your forms. A good place to start is the *forms documentation*

Handle uploaded files using the new API Replace use of uploaded files – that is, entries in `request.FILES` – as simple dictionaries with the new `UploadedFile`. The old dictionary syntax no longer works.

Thus, in a view like:

```
def my_view(request):
    f = request.FILES['file_field_name']
    ...
```

...you'd need to make the following changes:

Old (0.96)	New (1.0)
`f['content']`	`f.read()`
`f['filename']`	`f.name`
`f['content-type']`	`f.content_type`

Work with file fields using the new API The internal implementation of `django.db.models.FileField` have changed. A visible result of this is that the way you access special attributes (URL, filename, image size, etc) of these model fields has changed. You will need to make the following changes, assuming your model's `FileField` is called `myfile`:

Old (0.96)	New (1.0)
`myfile.get_content_filename()`	`myfile.content.path`
`myfile.get_content_url()`	`myfile.content.url`
`myfile.get_content_size()`	`myfile.content.size`
`myfile.save_content_file()`	`myfile.content.save()`
`myfile.get_content_width()`	`myfile.content.width`
`myfile.get_content_height()`	`myfile.content.height`

Note that the `width` and `height` attributes only make sense for `ImageField` fields. More details can be found in the *model API* documentation.

Use `Paginator` instead of `ObjectPaginator` The `ObjectPaginator` in 0.96 has been removed and replaced with an improved version, `django.core.paginator.Paginator`.

Templates

Learn to love autoescaping By default, the template system now automatically HTML-escapes the output of every variable. To learn more, see *Automatic HTML escaping*.

To disable auto-escaping for an individual variable, use the `safe` filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

To disable auto-escaping for an entire template, wrap the template (or just a particular section of the template) in the `autoescape` tag:

```
{% autoescape off %}
    ... unescaped template content here ...
{% endautoescape %}
```

Less-common changes The following changes are smaller, more localized changes. They should only affect more advanced users, but it's probably worth reading through the list and checking your code for these things.

Signals

- Add `**kwargs` to any registered signal handlers.
- Connect, disconnect, and send signals via methods on the `Signal` object instead of through module methods in `django.dispatch.dispatcher`.
- Remove any use of the `Anonymous` and `Any` sender options; they no longer exist. You can still receive signals sent by any sender by using `sender=None`
- Make any custom signals you've declared into instances of `django.dispatch.Signal` instead of anonymous objects.

Here's quick summary of the code changes you'll need to make:

Old (0.96)	New (1.0)
`def callback(sender)`	`def callback(sender, **kwargs)`
`sig = object()`	`sig = django.dispatch.Signal()`
`dispatcher.connect(callback, sig)`	`sig.connect(callback)`
`dispatcher.send(sig, sender)`	`sig.send(sender)`
`dispatcher.connect(callback, sig, sender=Any)`	`sig.connect(callback, sender=None)`

Comments If you were using Django 0.96's `django.contrib.comments` app, you'll need to upgrade to the new comments app introduced in 1.0. See *Upgrading from Django's previous comment system* for details.

Template tags

`spaceless` tag The spaceless template tag now removes *all* spaces between HTML tags, instead of preserving a single space.

Local flavors

U.S. local flavor `django.contrib.localflavor.usa` has been renamed to `django.contrib.localflavor.us`. This change was made to match the naming scheme of other local flavors. To migrate your code, all you need to do is change the imports.

Sessions

Getting a new session key `SessionBase.get_new_session_key()` has been renamed to `_get_new_session_key()`. `get_new_session_object()` no longer exists.

Fixtures

Loading a row no longer calls `save()` Previously, loading a row automatically ran the model's `save()` method. This is no longer the case, so any fields (for example: timestamps) that were auto-populated by a `save()` now need explicit values in any fixture.

Settings

Better exceptions The old `EnvironmentError` has split into an `ImportError` when Django fails to find the settings module and a `RuntimeError` when you try to reconfigure settings after having already used them

`LOGIN_URL` has moved The `LOGIN_URL` constant moved from `django.contrib.auth` into the `settings` module. Instead of using `from django.contrib.auth import LOGIN_URL` refer to `settings.LOGIN_URL`.

APPEND_SLASH behavior has been updated In 0.96, if a URL didn't end in a slash or have a period in the final component of its path, and APPEND_SLASH was True, Django would redirect to the same URL, but with a slash appended to the end. Now, Django checks to see whether the pattern without the trailing slash would be matched by something in your URL patterns. If so, no redirection takes place, because it is assumed you deliberately wanted to catch that pattern.

For most people, this won't require any changes. Some people, though, have URL patterns that look like this:

```
r'/some_prefix/(.*)$'
```

Previously, those patterns would have been redirected to have a trailing slash. If you always want a slash on such URLs, rewrite the pattern as:

```
r'/some_prefix/(.*/)$'
```

Smaller model changes

Different exception from get() Managers now return a MultipleObjectsReturned exception instead of AssertionError:

Old (0.96):

```
try:
    Model.objects.get(...)
except AssertionError:
    handle_the_error()
```

New (1.0):

```
try:
    Model.objects.get(...)
except Model.MultipleObjectsReturned:
    handle_the_error()
```

LazyDate has been fired The LazyDate helper class no longer exists.

Default field values and query arguments can both be callable objects, so instances of LazyDate can be replaced with a reference to datetime.datetime.now:

Old (0.96):

```
class Article(models.Model):
    title = models.CharField(maxlength=100)
    published = models.DateField(default=LazyDate())
```

New (1.0):

```
import datetime

class Article(models.Model):
    title = models.CharField(max_length=100)
    published = models.DateField(default=datetime.datetime.now)
```

DecimalField is new, and FloatField is now a proper float Old (0.96):

```
class MyModel(models.Model):
    field_name = models.FloatField(max_digits=10, decimal_places=3)
    ...
```

New (1.0):

```
class MyModel(models.Model):
    field_name = models.DecimalField(max_digits=10, decimal_places=3)
    ...
```

If you forget to make this change, you will see errors about `FloatField` not taking a `max_digits` attribute in `__init__`, because the new `FloatField` takes no precision-related arguments.

If you're using MySQL or PostgreSQL, no further changes are needed. The database column types for `DecimalField` are the same as for the old `FloatField`.

If you're using SQLite, you need to force the database to view the appropriate columns as decimal types, rather than floats. To do this, you'll need to reload your data. Do this after you have made the change to using `DecimalField` in your code and updated the Django code.

Warning: Back up your database first!
For SQLite, this means making a copy of the single file that stores the database (the name of that file is the `DATABASE_NAME` in your settings.py file).

To upgrade each application to use a `DecimalField`, you can do the following, replacing `<app>` in the code below with each app's name:

```
$ ./manage.py dumpdata --format=xml <app> > data-dump.xml
$ ./manage.py reset <app>
$ ./manage.py loaddata data-dump.xml
```

Notes:

1. It's important that you remember to use XML format in the first step of this process. We are exploiting a feature of the XML data dumps that makes porting floats to decimals with SQLite possible.
2. In the second step you will be asked to confirm that you are prepared to lose the data for the application(s) in question. Say yes; we'll restore this data in the third step, of course.
3. `DecimalField` is not used in any of the apps shipped with Django prior to this change being made, so you do not need to worry about performing this procedure for any of the standard Django models.

If something goes wrong in the above process, just copy your backed up database file over the original file and start again.

Internationalization

`django.views.i18n.set_language()` now requires a POST request Previously, a GET request was used. The old behavior meant that state (the locale used to display the site) could be changed by a GET request, which is against the HTTP specification's recommendations. Code calling this view must ensure that a POST request is now made, instead of a GET. This means you can no longer use a link to access the view, but must use a form submission of some kind (e.g. a button).

`_()` is no longer in builtins `_()` (the callable object whose name is a single underscore) is no longer monkey-patched into builtins – that is, it's no longer available magically in every module.

If you were previously relying on `_()` always being present, you should now explicitly import `ugettext` or `ugettext_lazy`, if appropriate, and alias it to _ yourself:

```
from django.utils.translation import ugettext as _
```

HTTP request/response objects

Dictionary access to `HttpRequest` `HttpRequest` objects no longer directly support dictionary-style access; previously, both `GET` and `POST` data were directly available on the `HttpRequest` object (e.g., you could check for a piece of form data by using `if 'some_form_key' in request` or by reading `request['some_form_key']`. This is no longer supported; if you need access to the combined `GET` and `POST` data, use `request.REQUEST` instead.

It is strongly suggested, however, that you always explicitly look in the appropriate dictionary for the type of request you expect to receive (`request.GET` or `request.POST`); relying on the combined `request.REQUEST` dictionary can mask the origin of incoming data.

Accessing `HTTPResponse` headers `django.http.HttpResponse.headers` has been renamed to `_headers` and `HttpResponse` now supports containment checking directly. So use `if header in response:` instead of `if header in response.headers:`.

Generic relations

Generic relations have been moved out of core The generic relation classes – `GenericForeignKey` and `GenericRelation` – have moved into the `django.contrib.contenttypes` module.

Testing

`django.test.Client.login()` has changed Old (0.96):

```
from django.test import Client
c = Client()
c.login('/path/to/login','myuser','mypassword')
```

New (1.0):

```
# ... same as above, but then:
c.login(username='myuser', password='mypassword')
```

Management commands

Running management commands from your code `django.core.management` has been greatly refactored.

Calls to management services in your code now need to use `call_command`. For example, if you have some test code that calls flush and load_data:

```
from django.core import management
management.flush(verbosity=0, interactive=False)
management.load_data(['test_data'], verbosity=0)
```

...you'll need to change this code to read:

```
from django.core import management
management.call_command('flush', verbosity=0, interactive=False)
management.call_command('loaddata', 'test_data', verbosity=0)
```

Subcommands must now precede options `django-admin.py` and `manage.py` now require subcommands to precede options. So:

```
$ django-admin.py --settings=foo.bar runserver
```

...no longer works and should be changed to:

```
$ django-admin.py runserver --settings=foo.bar
```

Syndication

`Feed.__init__` has changed The `__init__()` method of the syndication framework's `Feed` class now takes an `HttpRequest` object as its second parameter, instead of the feed's URL. This allows the syndication framework to work without requiring the sites framework. This only affects code that subclasses `Feed` and overrides the `__init__()` method, and code that calls `Feed.__init__()` directly.

Data structures

`SortedDictFromList` is gone `django.newforms.forms.SortedDictFromList` was removed. `django.utils.datastructures.SortedDict` can now be instantiated with a sequence of tuples.

To update your code:

1. Use `django.utils.datastructures.SortedDict` wherever you were using `django.newforms.forms.SortedDictFromList`.
2. Because `django.utils.datastructures.SortedDict.copy()` doesn't return a deepcopy as `SortedDictFromList.copy()` did, you will need to update your code if you were relying on a deepcopy. Do this by using `copy.deepcopy` directly.

Database backend functions

Database backend functions have been renamed Almost *all* of the database backend-level functions have been renamed and/or relocated. None of these were documented, but you'll need to change your code if you're using any of these functions, all of which are in `django.db`:

Old (0.96)	New (1.0)
`backend.get_autoinc_sql`	`connection.ops.autoinc_sql`
`backend.get_date_extract_sql`	`connection.ops.date_extract_sql`
`backend.get_date_trunc_sql`	`connection.ops.date_trunc_sql`
`backend.get_datetime_cast_sql`	`connection.ops.datetime_cast_sql`
`backend.get_deferrable_sql`	`connection.ops.deferrable_sql`
`backend.get_drop_foreignkey_sql`	`connection.ops.drop_foreignkey_sql`
`backend.get_fulltext_search_sql`	`connection.ops.fulltext_search_sql`

Continued on next page

Table 9.1 – continued from previous page

Old (0.96)	New (1.0)
backend.get_last_insert_id	connection.ops.last_insert_id
backend.get_limit_offset_sql	connection.ops.limit_offset_sql
backend.get_max_name_length	connection.ops.max_name_length
backend.get_pk_default_value	connection.ops.pk_default_value
backend.get_random_function_sql	connection.ops.random_function_sql
backend.get_sql_flush	connection.ops.sql_flush
backend.get_sql_sequence_reset	connection.ops.sequence_reset_sql
backend.get_start_transaction_sql	connection.ops.start_transaction_sql
backend.get_tablespace_sql	connection.ops.tablespace_sql
backend.quote_name	connection.ops.quote_name
backend.get_query_set_class	connection.ops.query_set_class
backend.get_field_cast_sql	connection.ops.field_cast_sql
backend.get_drop_sequence	connection.ops.drop_sequence_sql
backend.OPERATOR_MAPPING	connection.operators
backend.allows_group_by_ordinal	connection.features.allows_group_by_ordinal
backend.allows_unique_and_pk	connection.features.allows_unique_and_pk
backend.autoindexes_primary_keys	connection.features.autoindexes_primary_keys
backend.needs_datetime_string_cast	connection.features.needs_datetime_string_cast
backend.needs_upper_for_iops	connection.features.needs_upper_for_iops
backend.supports_constraints	connection.features.supports_constraints
backend.supports_tablespaces	connection.features.supports_tablespaces
backend.uses_case_insensitive_names	connection.features.uses_case_insensitive_names
backend.uses_custom_queryset	connection.features.uses_custom_queryset

A complete list of backwards-incompatible changes can be found at https://code.djangoproject.com/wiki/BackwardsIncompatibleChange

What's new in Django 1.0

A *lot*!

Since Django 0.96, we've made over 4,000 code commits, fixed more than 2,000 bugs, and edited, added, or removed around 350,000 lines of code. We've also added 40,000 lines of new documentation, and greatly improved what was already there.

In fact, new documentation is one of our favorite features of Django 1.0, so we might as well start there. First, there's a new documentation site:

- https://docs.djangoproject.com/

The documentation has been greatly improved, cleaned up, and generally made awesome. There's now dedicated search, indexes, and more.

We can't possibly document everything that's new in 1.0, but the documentation will be your definitive guide. Anywhere you see something like: New in version 1.0: This feature is new in Django 1.0 You'll know that you're looking at something new or changed.

The other major highlights of Django 1.0 are:

Re-factored admin application The Django administrative interface (`django.contrib.admin`) has been completely refactored; admin definitions are now completely decoupled from model definitions (no more `class Admin` declaration in models!), rewritten to use Django's new form-handling library (introduced in the 0.96 release as

`django.newforms`, and now available as simply `django.forms`) and redesigned with extensibility and customization in mind. Full documentation for the admin application is available online in the official Django documentation:

See the *admin reference* for details

Improved Unicode handling Django's internals have been refactored to use Unicode throughout; this drastically simplifies the task of dealing with non-Western-European content and data in Django. Additionally, utility functions have been provided to ease interoperability with third-party libraries and systems which may or may not handle Unicode gracefully. Details are available in Django's Unicode-handling documentation.

See *Unicode data*.

An improved ORM Django's object-relational mapper – the component which provides the mapping between Django model classes and your database, and which mediates your database queries – has been dramatically improved by a massive refactoring. For most users of Django this is backwards-compatible; the public-facing API for database querying underwent a few minor changes, but most of the updates took place in the ORM's internals. A guide to the changes, including backwards-incompatible modifications and mentions of new features opened up by this refactoring, is available on the Django wiki.

Automatic escaping of template variables To provide improved security against cross-site scripting (XSS) vulnerabilities, Django's template system now automatically escapes the output of variables. This behavior is configurable, and allows both variables and larger template constructs to be marked as safe (requiring no escaping) or unsafe (requiring escaping). A full guide to this feature is in the documentation for the `autoescape` tag.

`django.contrib.gis` (GeoDjango) A project over a year in the making, this adds world-class GIS (Geographic Information Systems) support to Django, in the form of a `contrib` application. Its documentation is currently being maintained externally, and will be merged into the main Django documentation shortly. Huge thanks go to Justin Bronn, Jeremy Dunck, Brett Hoerner and Travis Pinney for their efforts in creating and completing this feature.

See http://geodjango.org/ for details.

Pluggable file storage Django's built-in `FileField` and `ImageField` now can take advantage of pluggable file-storage backends, allowing extensive customization of where and how uploaded files get stored by Django. For details, see *the files documentation*; big thanks go to Marty Alchin for putting in the hard work to get this completed.

Jython compatibility Thanks to a lot of work from Leo Soto during a Google Summer of Code project, Django's codebase has been refactored to remove incompatibilities with Jython, an implementation of Python written in Java, which runs Python code on the Java Virtual Machine. Django is now compatible with the forthcoming Jython 2.5 release.

See *Running Django on Jython*.

Generic relations in forms and admin Classes are now included in `django.contrib.contenttypes` which can be used to support generic relations in both the admin interface and in end-user forms. See *the documentation for generic relations* for details.

`INSERT`/`UPDATE` distinction Although Django's default behavior of having a model's `save()` method automatically determine whether to perform an `INSERT` or an `UPDATE` at the SQL level is suitable for the majority of cases, there are occasional situations where forcing one or the other is useful. As a result, models can now support an additional parameter to `save()` which can force a specific operation.

See *Forcing an INSERT or UPDATE* for details.

Split `CacheMiddleware` Django's `CacheMiddleware` has been split into three classes: `CacheMiddleware` itself still exists and retains all of its previous functionality, but it is now built from two separate middleware classes which handle the two parts of caching (inserting into and reading from the cache) separately, offering additional flexibility for situations where combining these functions into a single middleware posed problems.

Full details, including updated notes on appropriate use, are in *the caching documentation*.

Refactored `django.contrib.comments` As part of a Google Summer of Code project, Thejaswi Puthraya carried out a major rewrite and refactoring of Django's bundled comment system, greatly increasing its flexibility and customizability. *Full documentation* is available, as well as *an upgrade guide* if you were using the previous incarnation of the comments application.

Removal of deprecated features A number of features and methods which had previously been marked as deprecated, and which were scheduled for removal prior to the 1.0 release, are no longer present in Django. These include imports of the form library from `django.newforms` (now located simply at `django.forms`), the `form_for_model` and `form_for_instance` helper functions (which have been replaced by `ModelForm`) and a number of deprecated features which were replaced by the dispatcher, file-uploading and file-storage refactorings introduced in the Django 1.0 alpha releases.

Known issues

We've done our best to make Django 1.0 as solid as possible, but unfortunately there are a couple of issues that we know about in the release.

Multi-table model inheritance with `to_field` If you're using *multiple table model inheritance*, be aware of this caveat: child models using a custom `parent_link` and `to_field` will cause database integrity errors. A set of models like the following are **not valid**:

```
class Parent(models.Model):
    name = models.CharField(max_length=10)
    other_value = models.IntegerField(unique=True)

class Child(Parent):
    father = models.OneToOneField(Parent, primary_key=True, to_field="other_value", parent_link=True)
    value = models.IntegerField()
```

This bug will be fixed in the next release of Django.

Caveats with support of certain databases Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and in particular many of the supported database differ greatly from version to version. It's a good idea to checkout our *notes on supported database*:

- *MySQL notes*
- *SQLite notes*
- *Oracle notes*

9.1.7 Pre-1.0 releases

Django version 0.96 release notes

Welcome to Django 0.96!

The primary goal for 0.96 is a cleanup and stabilization of the features introduced in 0.95. There have been a few small backwards-incompatible changes since 0.95, but the upgrade process should be fairly simple and should not require major changes to existing applications.

However, we're also releasing 0.96 now because we have a set of backwards-incompatible changes scheduled for the near future. Once completed, they will involve some code changes for application developers, so we recommend that you stick with Django 0.96 until the next official release; then you'll be able to upgrade in one step instead of needing to make incremental changes to keep up with the development version of Django.

Backwards-incompatible changes

The following changes may require you to update your code when you switch from 0.95 to 0.96:

`MySQLdb` version requirement Due to a bug in older versions of the `MySQLdb` Python module (which Django uses to connect to MySQL databases), Django's MySQL backend now requires version 1.2.1p2 or higher of `MySQLdb`, and will raise exceptions if you attempt to use an older version.

If you're currently unable to upgrade your copy of `MySQLdb` to meet this requirement, a separate, backwards-compatible backend, called "mysql_old", has been added to Django. To use this backend, change the `DATABASE_ENGINE` setting in your Django settings file from this:

```
DATABASE_ENGINE = "mysql"
```

to this:

```
DATABASE_ENGINE = "mysql_old"
```

However, we strongly encourage MySQL users to upgrade to a more recent version of `MySQLdb` as soon as possible, The "mysql_old" backend is provided only to ease this transition, and is considered deprecated; aside from any necessary security fixes, it will not be actively maintained, and it will be removed in a future release of Django.

Also, note that some features, like the new `DATABASE_OPTIONS` setting (see the *databases documentation* for details), are only available on the "mysql" backend, and will not be made available for "mysql_old".

Database constraint names changed The format of the constraint names Django generates for foreign key references have changed slightly. These names are generally only used when it is not possible to put the reference directly on the affected column, so they are not always visible.

The effect of this change is that running `manage.py reset` and similar commands against an existing database may generate SQL with the new form of constraint name, while the database itself contains constraints named in the old form; this will cause the database server to raise an error message about modifying non-existent constraints.

If you need to work around this, there are two methods available:

1. Redirect the output of `manage.py` to a file, and edit the generated SQL to use the correct constraint names before executing it.
2. Examine the output of `manage.py sqlall` to see the new-style constraint names, and use that as a guide to rename existing constraints in your database.

Name changes in `manage.py` A few of the options to `manage.py` have changed with the addition of fixture support:

- There are new `dumpdata` and `loaddata` commands which, as you might expect, will dump and load data to/from the database. These commands can operate against any of Django's supported serialization formats.
- The `sqlinitialdata` command has been renamed to `sqlcustom` to emphasize that `loaddata` should be used for data (and `sqlcustom` for other custom SQL – views, stored procedures, etc.).
- The vestigial `install` command has been removed. Use `syncdb`.

Backslash escaping changed The Django database API now escapes backslashes given as query parameters. If you have any database API code that matches backslashes, and it was working before (despite the lack of escaping), you'll have to change your code to "unescape" the slashes one level.

For example, this used to work:

```
# Find text containing a single backslash
MyModel.objects.filter(text__contains='\\\\')
```

The above is now incorrect, and should be rewritten as:

```
# Find text containing a single backslash
MyModel.objects.filter(text__contains='\\')
```

Removed ENABLE_PSYCO setting The `ENABLE_PSYCO` setting no longer exists. If your settings file includes `ENABLE_PSYCO` it will have no effect; to use Psyco, we recommend writing a middleware class to activate it.

What's new in 0.96?

This revision represents over a thousand source commits and over four hundred bug fixes, so we can't possibly catalog all the changes. Here, we describe the most notable changes in this release.

New forms library `django.newforms` is Django's new form-handling library. It's a replacement for `django.forms`, the old form/manipulator/validation framework. Both APIs are available in 0.96, but over the next two releases we plan to switch completely to the new forms system, and deprecate and remove the old system.

There are three elements to this transition:

- We've copied the current `django.forms` to `django.oldforms`. This allows you to upgrade your code *now* rather than waiting for the backwards-incompatible change and rushing to fix your code after the fact. Just change your import statements like this:

  ```
  from django import forms             # 0.95-style
  from django import oldforms as forms # 0.96-style
  ```

- The next official release of Django will move the current `django.newforms` to `django.forms`. This will be a backwards-incompatible change, and anyone still using the old version of `django.forms` at that time will need to change their import statements as described above.
- The next release after that will completely remove `django.oldforms`.

Although the `newforms` library will continue to evolve, it's ready for use for most common cases. We recommend that anyone new to form handling skip the old forms system and start with the new.

For more information about `django.newforms`, read the *newforms documentation*.

URLconf improvements You can now use any callable as the callback in URLconfs (previously, only strings that referred to callables were allowed). This allows a much more natural use of URLconfs. For example, this URLconf:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    ('^myview/$', 'mysite.myapp.views.myview')
)
```

can now be rewritten as:

```
from django.conf.urls.defaults import *
from mysite.myapp.views import myview

urlpatterns = patterns('',
    ('^myview/$', myview)
)
```

One useful application of this can be seen when using decorators; this change allows you to apply decorators to views *in your URLconf*. Thus, you can make a generic view require login very easily:

```
from django.conf.urls.defaults import *
from django.contrib.auth.decorators import login_required
from django.views.generic.list_detail import object_list
from mysite.myapp.models import MyModel

info = {
    "queryset" : MyModel.objects.all(),
}

urlpatterns = patterns('',
    ('^myview/$', login_required(object_list), info)
)
```

Note that both syntaxes (strings and callables) are valid, and will continue to be valid for the foreseeable future.

The test framework Django now includes a test framework so you can start transmuting fear into boredom (with apologies to Kent Beck). You can write tests based on `doctest` or `unittest` and test your views with a simple test client.

There is also new support for "fixtures" – initial data, stored in any of the supported *serialization formats*, that will be loaded into your database at the start of your tests. This makes testing with real data much easier.

See *the testing documentation* for the full details.

Improvements to the admin interface A small change, but a very nice one: dedicated views for adding and updating users have been added to the admin interface, so you no longer need to worry about working with hashed passwords in the admin.

Thanks

Since 0.95, a number of people have stepped forward and taken a major new role in Django's development. We'd like to thank these people for all their hard work:

- Russell Keith-Magee and Malcolm Tredinnick for their major code contributions. This release wouldn't have been possible without them.

- Our new release manager, James Bennett, for his work in getting out 0.95.1, 0.96, and (hopefully) future release.
- Our ticket managers Chris Beaven (aka SmileyChris), Simon Greenhill, Michael Radziej, and Gary Wilson. They agreed to take on the monumental task of wrangling our tickets into nicely cataloged submission. Figuring out what to work on is now about a million times easier; thanks again, guys.
- Everyone who submitted a bug report, patch or ticket comment. We can't possibly thank everyone by name – over 200 developers submitted patches that went into 0.96 – but everyone who's contributed to Django is listed in AUTHORS.

Django version 0.95 release notes

Welcome to the Django 0.95 release.

This represents a significant advance in Django development since the 0.91 release in January 2006. The details of every change in this release would be too extensive to list in full, but a summary is presented below.

Suitability and API stability

This release is intended to provide a stable reference point for developers wanting to work on production-level applications that use Django.

However, it's not the 1.0 release, and we'll be introducing further changes before 1.0. For a clear look at which areas of the framework will change (and which ones will *not* change) before 1.0, see the api-stability.txt file, which lives in the docs/ directory of the distribution.

You may have a need to use some of the features that are marked as "subject to API change" in that document, but that's OK with us as long as it's OK with you, and as long as you understand APIs may change in the future.

Fortunately, most of Django's core APIs won't be changing before version 1.0. There likely won't be as big of a change between 0.95 and 1.0 versions as there was between 0.91 and 0.95.

Changes and new features

The major changes in this release (for developers currently using the 0.91 release) are a result of merging the 'magic-removal' branch of development. This branch removed a number of constraints in the way Django code had to be written that were a consequence of decisions made in the early days of Django, prior to its open-source release. It's now possible to write more natural, Pythonic code that works as expected, and there's less "black magic" happening behind the scenes.

Aside from that, another main theme of this release is a dramatic increase in usability. We've made countless improvements in error messages, documentation, etc., to improve developers' quality of life.

The new features and changes introduced in 0.95 include:

- Django now uses a more consistent and natural filtering interface for retrieving objects from the database.
- User-defined models, functions and constants now appear in the module namespace they were defined in. (Previously everything was magically transferred to the django.models.* namespace.)
- Some optional applications, such as the FlatPage, Sites and Redirects apps, have been decoupled and moved into django.contrib. If you don't want to use these applications, you no longer have to install their database tables.
- Django now has support for managing database transactions.
- We've added the ability to write custom authentication and authorization backends for authenticating users against alternate systems, such as LDAP.

- We've made it easier to add custom table-level functions to models, through a new "Manager" API.
- It's now possible to use Django without a database. This simply means that the framework no longer requires you to have a working database set up just to serve dynamic pages. In other words, you can just use URLconfs/views on their own. Previously, the framework required that a database be configured, regardless of whether you actually used it.
- It's now more explicit and natural to override save() and delete() methods on models, rather than needing to hook into the pre_save() and post_save() method hooks.
- Individual pieces of the framework now can be configured without requiring the setting of an environment variable. This permits use of, for example, the Django templating system inside other applications.
- More and more parts of the framework have been internationalized, as we've expanded internationalization (i18n) support. The Django codebase, including code and templates, has now been translated, at least in part, into 31 languages. From Arabic to Chinese to Hungarian to Welsh, it is now possible to use Django's admin site in your native language.

The number of changes required to port from 0.91-compatible code to the 0.95 code base are significant in some cases. However, they are, for the most part, reasonably routine and only need to be done once. A list of the necessary changes is described in the Removing The Magic wiki page. There is also an easy checklist for reference when undertaking the porting operation.

Problem reports and getting help

Need help resolving a problem with Django? The documentation in the distribution is also available *online* at the Django Web site. The *FAQ* document is especially recommended, as it contains a number of issues that come up time and again.

For more personalized help, the django-users mailing list is a very active list, with more than 2,000 subscribers who can help you solve any sort of Django problem. We recommend you search the archives first, though, because many common questions appear with some regularity, and any particular problem may already have been answered.

Finally, for those who prefer the more immediate feedback offered by IRC, there's a #django channel on irc.freenode.net that is regularly populated by Django users and developers from around the world. Friendly people are usually available at any hour of the day – to help, or just to chat.

Thanks for using Django!

The Django Team July 2006

9.2 Development releases

These notes are retained for historical purposes. If you are upgrading between formal Django releases, you don't need to worry about these notes.

9.2.1 Django 1.4 beta release notes

February 15, 2012.

Welcome to Django 1.4 beta!

This is the second in a series of preview/development releases leading up to the eventual release of Django 1.4, scheduled for March 2012. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.4 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

Django 1.4 beta includes various new features and some minor backwards incompatible changes. There are also some features that have been dropped, which are detailed in *our deprecation plan*, and we've begun the deprecation process for some features.

Version numbering

Internally, Django's version number is represented by the tuple `django.VERSION`. This is used to generate human-readable version number strings; as of Django 1.4 beta 1, the algorithm for generating these strings has been changed to match the recommendations of **PEP 386**. This only affects the human-readable strings identifying Django alphas, betas and release candidates, and should not affect end users in any way.

For example purposes, the old algorithm would give Django 1.4 beta 1 a version string of the form "1.4 beta 1". The new algorithm generates the version string "1.4b1".

Python compatibility

While not a new feature, it's important to note that Django 1.4 introduces the second shift in our Python compatibility policy since Django's initial public debut. Django 1.2 dropped support for Python 2.3; now Django 1.4 drops support for Python 2.4. As such, the minimum Python version required for Django is now 2.5, and Django is tested and supported on Python 2.5, 2.6 and 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.5 or newer as their default version. If you're still using Python 2.4, however, you'll need to stick to Django 1.3 until you can upgrade; per *our support policy*, Django 1.3 will continue to receive security support until the release of Django 1.5.

Django does not support Python 3.x at this time. A document outlining our full timeline for deprecating Python 2.x and moving to Python 3.x will be published before the release of Django 1.4.

What's new in Django 1.4

Support for in-browser testing frameworks

Django 1.4 supports integration with in-browser testing frameworks like Selenium. The new `django.test.LiveServerTestCase` base class lets you test the interactions between your site's front and back ends more comprehensively. See the `documentation` for more details and concrete examples.

`SELECT FOR UPDATE` support

Django 1.4 now includes a `QuerySet.select_for_update()` method which generates a `SELECT ... FOR UPDATE` SQL query. This will lock rows until the end of the transaction, meaning that other transactions cannot modify or delete rows matched by a `FOR UPDATE` query.

For more details, see the documentation for `select_for_update()`.

`Model.objects.bulk_create` in the ORM

This method allows for more efficient creation of multiple objects in the ORM. It can provide significant performance increases if you have many objects. Django makes use of this internally, meaning some operations (such as database setup for test suites) have seen a performance benefit as a result.

See the `bulk_create()` docs for more information.

`QuerySet.prefetch_related`

Similar to `select_related()` but with a different strategy and broader scope, `prefetch_related()` has been added to `QuerySet`. This method returns a new `QuerySet` that will prefetch each of the specified related lookups in a single batch as soon as the query begins to be evaluated. Unlike `select_related`, it does the joins in Python, not in the database, and supports many-to-many relationships, `GenericForeignKey` and more. This allows you to fix a very common performance problem in which your code ends up doing O(n) database queries (or worse) if objects on your primary `QuerySet` each have many related objects that you also need.

Improved password hashing

Django's auth system (`django.contrib.auth`) stores passwords using a one-way algorithm. Django 1.3 uses the SHA1 algorithm, but increasing processor speeds and theoretical attacks have revealed that SHA1 isn't as secure as we'd like. Thus, Django 1.4 introduces a new password storage system: by default Django now uses the PBKDF2 algorithm (as recommended by NIST). You can also easily choose a different algorithm (including the popular bcrypt algorithm). For more details, see *How Django stores passwords*.

Warning: Django 1.4 alpha contained a bug that corrupted PBKDF2 hashes. To determine which accounts are affected, run `manage.py shell` and paste this snippet:

```
from base64 import b64decode
from django.contrib.auth.models import User
hash_len = {'pbkdf2_sha1': 20, 'pbkdf2_sha256': 32}
for user in User.objects.filter(password__startswith='pbkdf2_'):
    algo, _, _, hash = user.password.split('$')
    if len(b64decode(hash)) != hash_len[algo]:
        print user
```

These users should reset their passwords.

HTML5 Doctype

We've switched the admin and other bundled templates to use the HTML5 doctype. While Django will be careful to maintain compatibility with older browsers, this change means that you can use any HTML5 features you need in admin pages without having to lose HTML validity or override the provided templates to change the doctype.

List filters in admin interface

Prior to Django 1.4, the `admin` app allowed you to specify change list filters by specifying a field lookup, but didn't allow you to create custom filters. This has been rectified with a simple API (previously used internally and known as "FilterSpec"). For more details, see the documentation for `list_filter`.

Multiple sort in admin interface

The admin change list now supports sorting on multiple columns. It respects all elements of the `ordering` attribute, and sorting on multiple columns by clicking on headers is designed to mimic the behavior of desktop GUIs. The `get_ordering()` method for specifying the ordering dynamically (e.g. depending on the request) has also been added.

New `ModelAdmin` methods

A new `save_related()` method was added to `ModelAdmin` to ease customization of how related objects are saved in the admin.

Two other new methods, `get_list_display()` and `get_list_display_links()` were added to `ModelAdmin` to enable the dynamic customization of fields and links displayed on the admin change list.

Admin inlines respect user permissions

Admin inlines will now only allow those actions for which the user has permission. For `ManyToMany` relationships with an auto-created intermediate model (which does not have its own permissions), the change permission for the related model determines if the user has the permission to add, change or delete relationships.

Tools for cryptographic signing

Django 1.4 adds both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in Web applications.

See the *cryptographic signing* docs for more information.

Cookie-based session backend

Django 1.4 introduces a new cookie-based backend for the session framework which uses the tools for *cryptographic signing* to store the session data in the client's browser.

> **Warning:** Session data is signed and validated by the server, but is not encrypted. This means that a user can view any data stored in the session, but cannot change it. Please read the documentation for further clarification before using this backend.

See the *cookie-based session backend* docs for more information.

New form wizard

The previous `FormWizard` from the formtools contrib app has been replaced with a new implementation based on the class-based views introduced in Django 1.3. It features a pluggable storage API and doesn't require the wizard to pass around hidden fields for every previous step.

Django 1.4 ships with a session-based storage backend and a cookie-based storage backend. The latter uses the tools for *cryptographic signing* also introduced in Django 1.4 to store the wizard's state in the user's cookies.

See the *form wizard* docs for more information.

`reverse_lazy`

A lazily evaluated version of `django.core.urlresolvers.reverse()` was added to allow using URL reversals before the project's URLConf gets loaded.

Translating URL patterns

Django 1.4 gained the ability to look for a language prefix in the URL pattern when using the new `i18n_patterns()` helper function. Additionally, it's now possible to define translatable URL patterns using `ugettext_lazy()`. See *Internationalization: in URL patterns* for more information about the language prefix and how to internationalize URL patterns.

Contextual translation support for `{% trans %}` and `{% blocktrans %}`

The *contextual translation* support introduced in Django 1.3 via the `pgettext` function has been extended to the `trans` and `blocktrans` template tags using the new `context` keyword.

Customizable `SingleObjectMixin` URLConf kwargs

Two new attributes, `pk_url_kwarg` and `slug_url_kwarg`, have been added to `SingleObjectMixin` to enable the customization of URLConf keyword arguments used for single object generic views.

Assignment template tags

A new *assignment_tag* helper function was added to `template.Library` to ease the creation of template tags that store data in a specified context variable.

`*args` and `**kwargs` support for template tag helper functions

The *simple_tag*, *inclusion_tag* and newly introduced *assignment_tag* template helper functions may now accept any number of positional or keyword arguments. For example:

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then in the template any number of arguments may be passed to the template tag. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

No wrapping of exceptions in `TEMPLATE_DEBUG` mode

In previous versions of Django, whenever the `TEMPLATE_DEBUG` setting was `True`, any exception raised during template rendering (even exceptions unrelated to template syntax) were wrapped in `TemplateSyntaxError` and re-raised. This was done in order to provide detailed template source location information in the debug 500 page.

In Django 1.4, exceptions are no longer wrapped. Instead, the original exception is annotated with the source information. This means that catching exceptions from template rendering is now consistent regardless of the value of `TEMPLATE_DEBUG`, and there's no need to catch and unwrap `TemplateSyntaxError` in order to catch other errors.

`truncatechars` template filter

Added a filter which truncates a string to be no longer than the specified number of characters. Truncated strings end with a translatable ellipsis sequence ("..."). See the documentation for `truncatechars` for more details.

`static` template tag

The `staticfiles` contrib app has a new `static` template tag to refer to files saved with the `STATICFILES_STORAGE` storage backend. It uses the storage backend's `url` method and therefore supports advanced features such as *serving files from a cloud service*.

`CachedStaticFilesStorage` storage backend

In addition to the static template tag, the `staticfiles` contrib app now has a `CachedStaticFilesStorage` backend which caches the files it saves (when running the `collectstatic` management command) by appending the MD5 hash of the file's content to the filename. For example, the file `css/styles.css` would also be saved as `css/styles.55e7cbb9ba48.css`

See the `CachedStaticFilesStorage` docs for more information.

Simple clickjacking protection

We've added a middleware to provide easy protection against clickjacking using the `X-Frame-Options` header. It's not enabled by default for backwards compatibility reasons, but you'll almost certainly want to *enable it* to help plug that security hole for browsers that support the header.

CSRF improvements

We've made various improvements to our CSRF features, including the `ensure_csrf_cookie()` decorator which can help with AJAX heavy sites, protection for PUT and DELETE requests, and the `CSRF_COOKIE_SECURE` and `CSRF_COOKIE_PATH` settings which can improve the security and usefulness of the CSRF protection. See the *CSRF docs* for more information.

Error report filtering

Two new function decorators, `sensitive_variables()` and `sensitive_post_parameters()`, were added to allow designating the local variables and POST parameters which may contain sensitive information and should be filtered out of error reports.

All POST parameters are now systematically filtered out of error reports for certain views (`login`, `password_reset_confirm`, `password_change`, and `add_view` in `django.contrib.auth.views`, as well as `user_change_password` in the admin app) to prevent the leaking of sensitive information such as user passwords.

You may override or customize the default filtering by writing a *custom filter*. For more information see the docs on *Filtering error reports*.

Extended IPv6 support

The previously added support for IPv6 addresses when using the runserver management command in Django 1.3 has now been further extended by adding a `GenericIPAddressField` model field, a `GenericIPAddressField` form field and the validators `validate_ipv46_address` and `validate_ipv6_address`

Updated default project layout and `manage.py`

Django 1.4 ships with an updated default project layout and `manage.py` file for the `startproject` management command. These fix some issues with the previous `manage.py` handling of Python import paths that caused double imports, trouble moving from development to deployment, and other difficult-to-debug path issues.

The previous `manage.py` called functions that are now deprecated, and thus projects upgrading to Django 1.4 should update their `manage.py`. (The old-style `manage.py` will continue to work as before until Django 1.6; in 1.5 it will raise `DeprecationWarning`).

The new recommended `manage.py` file should look like this:

```
#!/usr/bin/env python
import os, sys

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "{{ project_name }}.settings")

    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

`{{ project_name }}` should be replaced with the Python package name of the actual project.

If settings, URLconfs, and apps within the project are imported or referenced using the project name prefix (e.g. `myproject.settings`, `ROOT_URLCONF = "myproject.urls"`, etc), the new `manage.py` will need to be moved one directory up, so it is outside the project package rather than adjacent to `settings.py` and `urls.py`.

For instance, with the following layout:

```
manage.py
mysite/
    __init__.py
    settings.py
    urls.py
    myapp/
        __init__.py
        models.py
```

You could import `mysite.settings`, `mysite.urls`, and `mysite.myapp`, but not `settings`, `urls`, or `myapp` as top-level modules.

Anything imported as a top-level module can be placed adjacent to the new `manage.py`. For instance, to decouple "myapp" from the project module and import it as just `myapp`, place it outside the `mysite/` directory:

```
manage.py
myapp/
    __init__.py
    models.py
mysite/
    __init__.py
    settings.py
    urls.py
```

If the same code is imported inconsistently (some places with the project prefix, some places without it), the imports will need to be cleaned up when switching to the new `manage.py`.

Improved WSGI support

The `startproject` management command now adds a `wsgi.py` module to the initial project layout, containing a simple WSGI application that can be used for *deploying with WSGI app servers*.

The `built-in development server` now supports using an externally-defined WSGI callable, so as to make it possible to run runserver with the same WSGI configuration that is used for deployment. A new `WSGI_APPLICATION` setting is available to configure which WSGI callable `runserver` uses.

(The `runfcgi` management command also internally wraps the WSGI callable configured via `WSGI_APPLICATION`.)

Custom project and app templates

The `startapp` and `startproject` management commands got a `--template` option for specifying a path or URL to a custom app or project template.

For example, Django will use the `/path/to/my_project_template` directory when running the following command:

```
django-admin.py startproject --template=/path/to/my_project_template myproject
```

You can also now provide a destination directory as the second argument to both `startapp` and `startproject`:

```
django-admin.py startapp myapp /path/to/new/app
django-admin.py startproject myproject /path/to/new/project
```

For more information, see the `startapp` and `startproject` documentation.

Support for time zones

Django 1.4 adds *support for time zones*. When it's enabled, Django stores date and time information in UTC in the database, uses time zone-aware datetime objects internally, and translates them to the end user's time zone in templates and forms.

Reasons for using this feature include:

- Customizing date and time display for users around the world.
- Storing datetimes in UTC for database portability and interoperability. (This argument doesn't apply to PostgreSQL, because it already stores timestamps with time zone information in Django 1.3.)
- Avoiding data corruption problems around DST transitions.

Time zone support is enabled by default in new projects created with `startproject`. If you want to use this feature in an existing project, there is a *migration guide*.

Two new date format strings

Two new `date` formats were added for use in template filters, template tags and *Format localization*:

- `e` – the name of the timezone of the given datetime object
- `o` – the ISO 8601 year number

Please make sure to update your *custom format files* if they contain either `e` or `o` in a format string. For example a Spanish localization format previously only escaped the `d` format character:

```
DATE_FORMAT = r'j \de F \de Y'
```

But now it needs to also escape `e` and `o`:

```
DATE_FORMAT = r'j \d\e F \d\e Y'
```

For more information, see the `date` documentation.

Minor features

Django 1.4 also includes several smaller improvements worth noting:

- A more usable stacktrace in the technical 500 page: frames in the stack trace which reference Django's code are dimmed out, while frames in user code are slightly emphasized. This change makes it easier to scan a stacktrace for issues in user code.
- *Tablespace support* in PostgreSQL.
- Customizable names for `simple_tag()`.
- In the documentation, a helpful *security overview* page.
- The `django.contrib.auth.models.check_password()` function has been moved to the `django.contrib.auth.utils` module. Importing it from the old location will still work, but you should update your imports.
- The `collectstatic` management command gained a `--clear` option to delete all files at the destination before copying or linking the static files.
- It is now possible to load fixtures containing forward references when using MySQL with the InnoDB database engine.
- A new 403 response handler has been added as `'django.views.defaults.permission_denied'`. You can set your own handler by setting the value of `django.conf.urls.handler403`. See the documentation about *the 403 (HTTP Forbidden) view* for more information.
- The `trans` template tag now takes an optional `as` argument to be able to retrieve a translation string without displaying it but setting a template context variable instead.
- The `if` template tag now supports `{% elif %}` clauses.
- A new plain text version of the HTTP 500 status code internal error page served when `DEBUG` is `True` is now sent to the client when Django detects that the request has originated in JavaScript code (`is_ajax()` is used for this).

  Similarly to its HTML counterpart, it contains a collection of different pieces of information about the state of the web application.

  This should make it easier to read when debugging interaction with client-side Javascript code.
- Added the `--no-location` option to the `makemessages` command.
- Changed the `locmem` cache backend to use `pickle.HIGHEST_PROTOCOL` for better compatibility with the other cache backends.
- Added support in the ORM for generating `SELECT` queries containing `DISTINCT ON`.

  The `distinct()` `QuerySet` method now accepts an optional list of model field names. If specified, then the `DISTINCT` statement is limited to these fields. This is only supported in PostgreSQL.

For more details, see the documentation for `distinct()`.

- New phrases added to `HIDDEN_SETTINGS` regex in django/views/debug.py.

  `'API'`, `'TOKEN'`, `'KEY'` were added, `'PASSWORD'` was changed to `'PASS'`.

Backwards incompatible changes in 1.4

django.contrib.admin

The included administration app `django.contrib.admin` has for a long time shipped with a default set of static files such as JavaScript, images and stylesheets. Django 1.3 added a new contrib app `django.contrib.staticfiles` to handle such files in a generic way and defined conventions for static files included in apps.

Starting in Django 1.4 the admin's static files also follow this convention to make it easier to deploy the included files. In previous versions of Django, it was also common to define an `ADMIN_MEDIA_PREFIX` setting to point to the URL where the admin's static files are served by a web server. This setting has now been deprecated and replaced by the more general setting `STATIC_URL`. Django will now expect to find the admin static files under the URL `<STATIC_URL>/admin/`.

If you've previously used a URL path for `ADMIN_MEDIA_PREFIX` (e.g. `/media/`) simply make sure `STATIC_URL` and `STATIC_ROOT` are configured and your web server serves the files correctly. The development server continues to serve the admin files just like before. Don't hesitate to consult the *static files howto* for further details.

In case your `ADMIN_MEDIA_PREFIX` is set to an specific domain (e.g. `http://media.example.com/admin/`) make sure to also set your `STATIC_URL` setting to the correct URL, for example `http://media.example.com/`.

> **Warning:** If you're implicitly relying on the path of the admin static files on your server's file system when you deploy your site, you have to update that path. The files were moved from `django/contrib/admin/media/` to `django/contrib/admin/static/admin/`.

Supported browsers for the admin

Django hasn't had a clear policy on which browsers are supported for using the admin app. Django's new policy formalizes existing practices: YUI's A-grade browsers should provide a fully-functional admin experience, with the notable exception of IE6, which is no longer supported.

Released over ten years ago, IE6 imposes many limitations on modern web development. The practical implications of this policy are that contributors are free to improve the admin without consideration for these limitations.

This new policy **has no impact** on development outside of the admin. Users of Django are free to develop webapps compatible with any range of browsers.

Removed admin icons

As part of an effort to improve the performance and usability of the admin's changelist sorting interface and of the admin's `horizontal` and `vertical` "filter" widgets, some icon files were removed and grouped into two sprite files.

Specifically: `selector-add.gif`, `selector-addall.gif`, `selector-remove.gif`, `selector-removeall.gif`, `selector_stacked-add.gif` and `selector_stacked-remove.gif`

were combined into `selector-icons.gif`; and `arrow-up.gif` and `arrow-down.gif` were combined into `sorting-icons.gif`.

If you used those icons to customize the admin then you will want to replace them with your own icons or retrieve them from a previous release.

CSS class names in admin forms

To avoid conflicts with other common CSS class names (e.g. "button"), a prefix "field-" has been added to all CSS class names automatically generated from the form field names in the main admin forms, stacked inline forms and tabular inline cells. You will need to take that prefix into account in your custom style sheets or javascript files if you previously used plain field names as selectors for custom styles or javascript transformations.

Compatibility with old signed data

Django 1.3 changed the cryptographic signing mechanisms used in a number of places in Django. While Django 1.3 kept fallbacks that would accept hashes produced by the previous methods, these fallbacks are removed in Django 1.4.

So, if you upgrade to Django 1.4 directly from 1.2 or earlier, you may lose/invalidate certain pieces of data that have been cryptographically signed using an old method. To avoid this, use Django 1.3 first for a period of time to allow the signed data to expire naturally. The affected parts are detailed below, with 1) the consequences of ignoring this advice and 2) the amount of time you need to run Django 1.3 for the data to expire or become irrelevant.

- `contrib.sessions` data integrity check
  - consequences: the user will be logged out, and session data will be lost.
  - time period: defined by `SESSION_COOKIE_AGE`.
- `contrib.auth` password reset hash
  - consequences: password reset links from before the upgrade will not work.
  - time period: defined by `PASSWORD_RESET_TIMEOUT_DAYS`.

Form-related hashes — these are much shorter lifetime, and are relevant only for the short window where a user might fill in a form generated by the pre-upgrade Django instance, and try to submit it to the upgraded Django instance:

- `contrib.comments` form security hash
  - consequences: the user will see a validation error "Security hash failed".
  - time period: the amount of time you expect users to take filling out comment forms.
- `FormWizard` security hash
  - consequences: the user will see an error about the form having expired, and will be sent back to the first page of the wizard, losing the data they have entered so far.
  - time period: the amount of time you expect users to take filling out the affected forms.
- CSRF check
  - Note: This is actually a Django 1.1 fallback, not Django 1.2, and applies only if you are upgrading from 1.1.
  - consequences: the user will see a 403 error with any CSRF protected POST form.
  - time period: the amount of time you expect user to take filling out such forms.

django.contrib.flatpages

Starting in the 1.4 release the `FlatpageFallbackMiddleware` only adds a trailing slash and redirects if the resulting URL refers to an existing flatpage. For example, requesting `/notaflatpageoravalidurl` in a previous version would redirect to `/notaflatpageoravalidurl/`, which would subsequently raise a 404. Requesting `/notaflatpageoravalidurl` now will immediately raise a 404. Additionally redirects returned by flatpages are now permanent (301 status code) to match the behavior of the `CommonMiddleware`.

Serialization of `datetime` and `time`

As a consequence of time zone support, and according to the ECMA-262 specification, some changes were made to the JSON serializer:

- It includes the time zone for aware datetime objects. It raises an exception for aware time objects.
- It includes milliseconds for datetime and time objects. There is still some precision loss, because Python stores microseconds (6 digits) and JSON only supports milliseconds (3 digits). However, it's better than discarding microseconds entirely.

The XML serializer was also changed to use the ISO8601 format for datetimes. The letter `T` is used to separate the date part from the time part, instead of a space. Time zone information is included in the `[+-]HH:MM` format.

The serializers will dump datetimes in fixtures with these new formats. They can still load fixtures that use the old format.

`supports_timezone` changed to `False` for SQLite

The database feature `supports_timezone` used to be `True` for SQLite. Indeed, if you saved an aware datetime object, SQLite stored a string that included an UTC offset. However, this offset was ignored when loading the value back from the database, which could corrupt the data.

In the context of time zone support, this flag was changed to `False`, and datetimes are now stored without time zone information in SQLite. When `USE_TZ` is `False`, if you attempt to save an aware datetime object, Django raises an exception.

Database connection's thread-locality

`DatabaseWrapper` objects (i.e. the connection objects referenced by `django.db.connection` and `django.db.connections["some_alias"]`) used to be thread-local. They are now global objects in order to be potentially shared between multiple threads. While the individual connection objects are now global, the `django.db.connections` dictionary referencing those objects is still thread-local. Therefore if you just use the ORM or `DatabaseWrapper.cursor()` then the behavior is still the same as before. Note, however, that `django.db.connection` does not directly reference the default `DatabaseWrapper` object anymore and is now a proxy to access that object's attributes. If you need to access the actual `DatabaseWrapper` object, use `django.db.connections[DEFAULT_DB_ALIAS]` instead.

As part of this change, all underlying SQLite connections are now enabled for potential thread-sharing (by passing the `check_same_thread=False` attribute to pysqlite). `DatabaseWrapper` however preserves the previous behavior by disabling thread-sharing by default, so this does not affect any existing code that purely relies on the ORM or on `DatabaseWrapper.cursor()`.

Finally, while it is now possible to pass connections between threads, Django does not make any effort to synchronize access to the underlying backend. Concurrency behavior is defined by the underlying backend implementation. Check their documentation for details.

COMMENTS_BANNED_USERS_GROUP setting

Django's *comments app* has historically supported excluding the comments of a special user group, but we've never documented the feature properly and didn't enforce the exclusion in other parts of the app such as the template tags. To fix this problem, we removed the code from the feed class.

If you rely on the feature and want to restore the old behavior, simply use a custom comment model manager to exclude the user group, like this:

```
from django.conf import settings
from django.contrib.comments.managers import CommentManager

class BanningCommentManager(CommentManager):
    def get_query_set(self):
        qs = super(BanningCommentManager, self).get_query_set()
        if getattr(settings, 'COMMENTS_BANNED_USERS_GROUP', None):
            where = ['user_id NOT IN (SELECT user_id FROM auth_user_groups WHERE group_id = %s)']
            params = [settings.COMMENTS_BANNED_USERS_GROUP]
            qs = qs.extra(where=where, params=params)
        return qs
```

Save this model manager in your custom comment app (e.g. in `my_comments_app/managers.py`) and add it your *custom comment app model*:

```
from django.db import models
from django.contrib.comments.models import Comment

from my_comments_app.managers import BanningCommentManager

class CommentWithTitle(Comment):
    title = models.CharField(max_length=300)

    objects = BanningCommentManager()
```

For more details, see the documentation about *customizing the comments framework*.

IGNORABLE_404_STARTS and *IGNORABLE_404_ENDS* settings

Until Django 1.3, it was possible to exclude some URLs from Django's *404 error reporting* by adding prefixes to `IGNORABLE_404_STARTS` and suffixes to `IGNORABLE_404_ENDS`.

In Django 1.4, these two settings are superseded by `IGNORABLE_404_URLS`, which is a list of compiled regular expressions. Django won't send an email for 404 errors on URLs that match any of them.

Furthermore, the previous settings had some rather arbitrary default values:

```
IGNORABLE_404_STARTS = ('/cgi-bin/', '/_vti_bin', '/_vti_inf')
IGNORABLE_404_ENDS = ('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi',
                      'favicon.ico', '.php')
```

It's not Django's role to decide if your website has a legacy `/cgi-bin/` section or a `favicon.ico`. As a consequence, the default values of `IGNORABLE_404_URLS`, `IGNORABLE_404_STARTS` and `IGNORABLE_404_ENDS` are all now empty.

If you have customized `IGNORABLE_404_STARTS` or `IGNORABLE_404_ENDS`, or if you want to keep the old default value, you should add the following lines in your settings file:

```
import re
IGNORABLE_404_URLS = (
    # for each <prefix> in IGNORABLE_404_STARTS
    re.compile(r'^<prefix>'),
    # for each <suffix> in IGNORABLE_404_ENDS
    re.compile(r'<suffix>$'),
)
```

Don't forget to escape characters that have a special meaning in a regular expression.

CSRF protection extended to PUT and DELETE

Previously, Django's *CSRF protection* provided protection against only POST requests. Since use of PUT and DELETE methods in AJAX applications is becoming more common, we now protect all methods not defined as safe by **RFC 2616** i.e. we exempt GET, HEAD, OPTIONS and TRACE, and enforce protection on everything else.

If you are using PUT or DELETE methods in AJAX applications, please see the *instructions about using AJAX and CSRF*.

`django.core.template_loaders`

This was an alias to `django.template.loader` since 2005, it has been removed without emitting a warning due to the length of the deprecation. If your code still referenced this please use `django.template.loader` instead.

`django.db.models.fields.URLField.verify_exists`

This functionality has been removed due to intractable performance and security issues. Any existing usage of `verify_exists` should be removed.

`django.core.files.storage.Storage.open`

The `open` method of the base Storage class took an obscure parameter `mixin` which allowed you to dynamically change the base classes of the returned file object. This has been removed. In the rare case you relied on the *mixin* parameter, you can easily achieve the same by overriding the *open* method, e.g.:

```
from django.core.files import File
from django.core.files.storage import FileSystemStorage

class Spam(File):
    """
    Spam, spam, spam, spam and spam.
    """
    def ham(self):
        return 'eggs'

class SpamStorage(FileSystemStorage):
    """
    A custom file storage backend.
    """
    def open(self, name, mode='rb'):
        return Spam(open(self.path(name), mode))
```

YAML deserializer now uses `yaml.safe_load`

`yaml.load` is able to construct any Python object, which may trigger arbitrary code execution if you process a YAML document that comes from an untrusted source. This feature isn't necessary for Django's YAML deserializer, whose primary use is to load fixtures consisting of simple objects. Even though fixtures are trusted data, for additional security, the YAML deserializer now uses `yaml.safe_load`.

Features deprecated in 1.4

Old styles of calling `cache_page` decorator

Some legacy ways of calling `cache_page()` have been deprecated, please see the docs for the correct way to use this decorator.

Support for PostgreSQL versions older than 8.2

Django 1.3 dropped support for PostgreSQL versions older than 8.0 and the relevant documents suggested to use a recent version because of performance reasons but more importantly because end of the upstream support periods for releases 8.0 and 8.1 was near (November 2010).

Django 1.4 takes that policy further and sets 8.2 as the minimum PostgreSQL version it officially supports.

Request exceptions are now always logged

When *logging support* was added to Django in 1.3, the admin error email support was moved into the `django.utils.log.AdminEmailHandler`, attached to the `'django.request'` logger. In order to maintain the established behavior of error emails, the `'django.request'` logger was called only when `DEBUG` was `False`.

To increase the flexibility of error logging for requests, the `'django.request'` logger is now called regardless of the value of `DEBUG`, and the default settings file for new projects now includes a separate filter attached to `django.utils.log.AdminEmailHandler` to prevent admin error emails in `DEBUG` mode:

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse'
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

If your project was created prior to this change, your `LOGGING` setting will not include this new filter. In order to maintain backwards-compatibility, Django will detect that your `'mail_admins'` handler configuration includes no `'filters'` section, and will automatically add this filter for you and issue a pending-deprecation warning. This will become a deprecation warning in Django 1.5, and in Django 1.6 the backwards-compatibility shim will be removed entirely.

The existence of any `'filters'` key under the `'mail_admins'` handler will disable this backward-compatibility shim and deprecation warning.

django.conf.urls.defaults

Until Django 1.3 the functions `include()`, `patterns()` and `url()` plus `handler404`, `handler500` were located in a `django.conf.urls.defaults` module.

Starting with Django 1.4 they are now available in `django.conf.urls`.

django.contrib.databrowse

Databrowse has not seen active development for some time, and this does not show any sign of changing. There had been a suggestion for a GSOC project to integrate the functionality of databrowse into the admin, but no progress was made. While Databrowse has been deprecated, an enhancement of `django.contrib.admin` providing a similar feature set is still possible.

The code that powers Databrowse is licensed under the same terms as Django itself, and so is available to be adopted by an individual or group as a third-party project.

django.core.management.setup_environ

This function temporarily modified `sys.path` in order to make the parent "project" directory importable under the old flat `startproject` layout. This function is now deprecated, as its path workarounds are no longer needed with the new `manage.py` and default project layout.

This function was never documented or part of the public API, but was widely recommended for use in setting up a "Django environment" for a user script. These uses should be replaced by setting the `DJANGO_SETTINGS_MODULE` environment variable or using `django.conf.settings.configure()`.

django.core.management.execute_manager

This function was previously used by `manage.py` to execute a management command. It is identical to `django.core.management.execute_from_command_line`, except that it first calls `setup_environ`, which is now deprecated. As such, `execute_manager` is also deprecated; `execute_from_command_line` can be used instead. Neither of these functions is documented as part of the public API, but a deprecation path is needed due to use in existing `manage.py` files.

is_safe and needs_autoescape attributes of template filters

Two flags, `is_safe` and `needs_autoescape`, define how each template filter interacts with Django's auto-escaping behavior. They used to be attributes of the filter function:

```
@register.filter
def noop(value):
    return value
noop.is_safe = True
```

However, this technique caused some problems in combination with decorators, especially `@stringfilter`. Now, the flags are keyword arguments of `@register.filter`:

```
@register.filter(is_safe=True)
def noop(value):
    return value
```

See *filters and auto-escaping* for more information.

Session cookies now have the `httponly` flag by default

Session cookies now include the `httponly` attribute by default to help reduce the impact of potential XSS attacks. As a consequence of this change, session cookie data, including sessionid, is no longer accessible from Javascript in many browsers. For strict backwards compatibility, use `SESSION_COOKIE_HTTPONLY = False` in your settings file.

Wildcard expansion of application names in *INSTALLED_APPS*

Until Django 1.3, `INSTALLED_APPS` accepted wildcards in application names, like `django.contrib.*`. The expansion was performed by a filesystem-based implementation of `from <package> import *`. Unfortunately, this can't be done reliably.

This behavior was never documented. Since it is un-pythonic and not obviously useful, it was removed in Django 1.4. If you relied on it, you must edit your settings file to list all your applications explicitly.

`HttpRequest.raw_post_data` renamed to `HttpRequest.body`

This attribute was confusingly named `HttpRequest.raw_post_data`, but it actually provided the body of the HTTP request. It's been renamed to `HttpRequest.body`, and `HttpRequest.raw_post_data` has been deprecated.

The Django 1.4 roadmap

Before the final Django 1.4 release, several other preview/development releases will be made available. The current schedule consists of at least the following:

- Week of **January 13, 2012**: First Django 1.4 beta release; final feature freeze for Django 1.4.
- Week of **February 27, 2012**: First Django 1.4 release candidate; string freeze for translations.
- Week of **March 5, 2012**: Django 1.4 final release.

If necessary, additional alpha, beta or release-candidate packages will be issued prior to the final 1.4 release. Django 1.4 will be released approximately one week after the final release candidate.

What you can do to help

In order to provide a high-quality 1.4 release, we need your help. Although this beta release is, again, *not* intended for production use, you can help the Django team by trying out the beta codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.4 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Several development sprints will also be taking place before the 1.4 release; these will typically be announced in advance on the django-developers mailing list, and anyone who wants to help is welcome to join in.

9.2.2 Django 1.4 alpha release notes

December 22, 2011.

Welcome to Django 1.4 alpha!

This is the first in a series of preview/development releases leading up to the eventual release of Django 1.4, scheduled for March 2012. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.4 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

Django 1.4 alpha includes various new features and some minor backwards incompatible changes. There are also some features that have been dropped, which are detailed in *our deprecation plan*, and we've begun the deprecation process for some features.

Python compatibility

While not a new feature, it's important to note that Django 1.4 introduces the second shift in our Python compatibility policy since Django's initial public debut. Django 1.2 dropped support for Python 2.3; now Django 1.4 drops support for Python 2.4. As such, the minimum Python version required for Django is now 2.5, and Django is tested and supported on Python 2.5, 2.6 and 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.5 or newer as their default version. If you're still using Python 2.4, however, you'll need to stick to Django 1.3 until you can upgrade; per *our support policy*, Django 1.3 will continue to receive security support until the release of Django 1.5.

Django does not support Python 3.x at this time. A document outlining our full timeline for deprecating Python 2.x and moving to Python 3.x will be published before the release of Django 1.4.

What's new in Django 1.4

Support for in-browser testing frameworks

Django 1.4 supports integration with in-browser testing frameworks like Selenium. The new `django.test.LiveServerTestCase` base class lets you test the interactions between your site's front and back ends more comprehensively. See the `documentation` for more details and concrete examples.

`SELECT FOR UPDATE` support

Django 1.4 now includes a `QuerySet.select_for_update()` method which generates a `SELECT ... FOR UPDATE` SQL query. This will lock rows until the end of the transaction, meaning that other transactions cannot modify or delete rows matched by a `FOR UPDATE` query.

For more details, see the documentation for `select_for_update()`.

`Model.objects.bulk_create` in the ORM

This method allows for more efficient creation of multiple objects in the ORM. It can provide significant performance increases if you have many objects. Django makes use of this internally, meaning some operations (such as database setup for test suites) have seen a performance benefit as a result.

See the `bulk_create()` docs for more information.

`QuerySet.prefetch_related`

Similar to `select_related()` but with a different strategy and broader scope, `prefetch_related()` has been added to `QuerySet`. This method returns a new `QuerySet` that will prefetch each of the specified related lookups in a single batch as soon as the query begins to be evaluated. Unlike `select_related`, it does the joins in Python, not in the database, and supports many-to-many relationships, `GenericForeignKey` and more. This allows you to fix a very common performance problem in which your code ends up doing O(n) database queries (or worse) if objects on your primary `QuerySet` each have many related objects that you also need.

Improved password hashing

Django's auth system (`django.contrib.auth`) stores passwords using a one-way algorithm. Django 1.3 uses the SHA1 algorithm, but increasing processor speeds and theoretical attacks have revealed that SHA1 isn't as secure as we'd like. Thus, Django 1.4 introduces a new password storage system: by default Django now uses the PBKDF2 algorithm (as recommended by NIST). You can also easily choose a different algorithm (including the popular bcrypt algorithm). For more details, see *How Django stores passwords*.

HTML5 Doctype

We've switched the admin and other bundled templates to use the HTML5 doctype. While Django will be careful to maintain compatibility with older browsers, this change means that you can use any HTML5 features you need in admin pages without having to lose HTML validity or override the provided templates to change the doctype.

List filters in admin interface

Prior to Django 1.4, the `admin` app allowed you to specify change list filters by specifying a field lookup, but didn't allow you to create custom filters. This has been rectified with a simple API (previously used internally and known as "FilterSpec"). For more details, see the documentation for `list_filter`.

Multiple sort in admin interface

The admin change list now supports sorting on multiple columns. It respects all elements of the `ordering` attribute, and sorting on multiple columns by clicking on headers is designed to mimic the behavior of desktop GUIs. The `get_ordering()` method for specifying the ordering dynamically (e.g. depending on the request) has also been added.

New `ModelAdmin` methods

A new `save_related()` method was added to `ModelAdmin` to ease customization of how related objects are saved in the admin.

Two other new methods, `get_list_display()` and `get_list_display_links()` were added to `ModelAdmin` to enable the dynamic customization of fields and links displayed on the admin change list.

Admin inlines respect user permissions

Admin inlines will now only allow those actions for which the user has permission. For `ManyToMany` relationships with an auto-created intermediate model (which does not have its own permissions), the change permission for the related model determines if the user has the permission to add, change or delete relationships.

Tools for cryptographic signing

Django 1.4 adds both a low-level API for signing values and a high-level API for setting and reading signed cookies, one of the most common uses of signing in Web applications.

See the *cryptographic signing* docs for more information.

Cookie-based session backend

Django 1.4 introduces a new cookie-based backend for the session framework which uses the tools for *cryptographic signing* to store the session data in the client's browser.

See the *cookie-based session backend* docs for more information.

New form wizard

The previous `FormWizard` from the formtools contrib app has been replaced with a new implementation based on the class-based views introduced in Django 1.3. It features a pluggable storage API and doesn't require the wizard to pass around hidden fields for every previous step.

Django 1.4 ships with a session-based storage backend and a cookie-based storage backend. The latter uses the tools for *cryptographic signing* also introduced in Django 1.4 to store the wizard's state in the user's cookies.

See the *form wizard* docs for more information.

`reverse_lazy`

A lazily evaluated version of `django.core.urlresolvers.reverse()` was added to allow using URL reversals before the project's URLConf gets loaded.

Translating URL patterns

Django 1.4 gained the ability to look for a language prefix in the URL pattern when using the new `i18n_patterns()` helper function. Additionally, it's now possible to define translatable URL patterns using `ugettext_lazy()`. See *Internationalization: in URL patterns* for more information about the language prefix and how to internationalize URL patterns.

Contextual translation support for `{% trans %}` and `{% blocktrans %}`

The *contextual translation* support introduced in Django 1.3 via the `pgettext` function has been extended to the `trans` and `blocktrans` template tags using the new `context` keyword.

Customizable `SingleObjectMixin` URLConf kwargs

Two new attributes, `pk_url_kwarg` and `slug_url_kwarg`, have been added to `SingleObjectMixin` to enable the customization of URLConf keyword arguments used for single object generic views.

Assignment template tags

A new *assignment_tag* helper function was added to `template.Library` to ease the creation of template tags that store data in a specified context variable.

`*args` and `**kwargs` support for template tag helper functions

The *simple_tag*, *inclusion_tag* and newly introduced *assignment_tag* template helper functions may now accept any number of positional or keyword arguments. For example:

```
@register.simple_tag
def my_tag(a, b, *args, **kwargs):
    warning = kwargs['warning']
    profile = kwargs['profile']
    ...
    return ...
```

Then in the template any number of arguments may be passed to the template tag. For example:

```
{% my_tag 123 "abcd" book.title warning=message|lower profile=user.profile %}
```

No wrapping of exceptions in `TEMPLATE_DEBUG` mode

In previous versions of Django, whenever the `TEMPLATE_DEBUG` setting was `True`, any exception raised during template rendering (even exceptions unrelated to template syntax) were wrapped in `TemplateSyntaxError` and re-raised. This was done in order to provide detailed template source location information in the debug 500 page.

In Django 1.4, exceptions are no longer wrapped. Instead, the original exception is annotated with the source information. This means that catching exceptions from template rendering is now consistent regardless of the value of `TEMPLATE_DEBUG`, and there's no need to catch and unwrap `TemplateSyntaxError` in order to catch other errors.

`truncatechars` template filter

Added a filter which truncates a string to be no longer than the specified number of characters. Truncated strings end with a translatable ellipsis sequence ("..."). See the documentation for `truncatechars` for more details.

`static` template tag

The `staticfiles` contrib app has a new `static` template tag to refer to files saved with the `STATICFILES_STORAGE` storage backend. It uses the storage backend's `url` method and therefore supports advanced features such as *serving files from a cloud service*.

`CachedStaticFilesStorage` storage backend

In addition to the static template tag, the `staticfiles` contrib app now has a `CachedStaticFilesStorage` backend which caches the files it saves (when running the `collectstatic` management command) by appending the MD5 hash of the file's content to the filename. For example, the file `css/styles.css` would also be saved as `css/styles.55e7cbb9ba48.css`

See the `CachedStaticFilesStorage` docs for more information.

Simple clickjacking protection

We've added a middleware to provide easy protection against clickjacking using the `X-Frame-Options` header. It's not enabled by default for backwards compatibility reasons, but you'll almost certainly want to *enable it* to help plug that security hole for browsers that support the header.

CSRF improvements

We've made various improvements to our CSRF features, including the `ensure_csrf_cookie()` decorator which can help with AJAX heavy sites, protection for PUT and DELETE requests, and the `CSRF_COOKIE_SECURE` and `CSRF_COOKIE_PATH` settings which can improve the security and usefulness of the CSRF protection. See the *CSRF docs* for more information.

Error report filtering

Two new function decorators, `sensitive_variables()` and `sensitive_post_parameters()`, were added to allow designating the local variables and POST parameters which may contain sensitive information and should be filtered out of error reports.

All POST parameters are now systematically filtered out of error reports for certain views (`login`, `password_reset_confirm`, `password_change`, and `add_view` in `django.contrib.auth.views`, as well as `user_change_password` in the admin app) to prevent the leaking of sensitive information such as user passwords.

You may override or customize the default filtering by writing a *custom filter*. For more information see the docs on *Filtering error reports*.

Extended IPv6 support

The previously added support for IPv6 addresses when using the runserver management command in Django 1.3 has now been further extended by adding a `GenericIPAddressField` model field, a `GenericIPAddressField` form field and the validators `validate_ipv46_address` and `validate_ipv6_address`

Updated default project layout and `manage.py`

Django 1.4 ships with an updated default project layout and `manage.py` file for the `startproject` management command. These fix some issues with the previous `manage.py` handling of Python import paths that caused double imports, trouble moving from development to deployment, and other difficult-to-debug path issues.

The previous `manage.py` called functions that are now deprecated, and thus projects upgrading to Django 1.4 should update their `manage.py`. (The old-style `manage.py` will continue to work as before until Django 1.6; in 1.5 it will raise `DeprecationWarning`).

The new recommended `manage.py` file should look like this:

```
#!/usr/bin/env python
import os, sys

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "{{ project_name }}.settings")

    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

`{{ project_name }}` should be replaced with the Python package name of the actual project.

If settings, URLconfs, and apps within the project are imported or referenced using the project name prefix (e.g. `myproject.settings`, `ROOT_URLCONF = "myproject.urls"`, etc), the new `manage.py` will need to be moved one directory up, so it is outside the project package rather than adjacent to `settings.py` and `urls.py`.

For instance, with the following layout:

```
manage.py
mysite/
    __init__.py
    settings.py
    urls.py
    myapp/
        __init__.py
        models.py
```

You could import `mysite.settings`, `mysite.urls`, and `mysite.myapp`, but not `settings`, `urls`, or `myapp` as top-level modules.

Anything imported as a top-level module can be placed adjacent to the new `manage.py`. For instance, to decouple "myapp" from the project module and import it as just `myapp`, place it outside the `mysite/` directory:

```
manage.py
myapp/
    __init__.py
    models.py
mysite/
    __init__.py
    settings.py
    urls.py
```

If the same code is imported inconsistently (some places with the project prefix, some places without it), the imports will need to be cleaned up when switching to the new `manage.py`.

Improved WSGI support

The `startproject` management command now adds a `wsgi.py` module to the initial project layout, containing a simple WSGI application that can be used for *deploying with WSGI app servers*.

The `built-in development server` now supports using an externally-defined WSGI callable, so as to make it possible to run runserver with the same WSGI configuration that is used for deployment. A new `WSGI_APPLICATION` setting is available to configure which WSGI callable `runserver` uses.

(The `runfcgi` management command also internally wraps the WSGI callable configured via `WSGI_APPLICATION`.)

Custom project and app templates

The `startapp` and `startproject` management commands got a `--template` option for specifying a path or URL to a custom app or project template.

For example, Django will use the `/path/to/my_project_template` directory when running the following command:

```
django-admin.py startproject --template=/path/to/my_project_template myproject
```

You can also now provide a destination directory as the second argument to both `startapp` and `startproject`:

```
django-admin.py startapp myapp /path/to/new/app
django-admin.py startproject myproject /path/to/new/project
```

For more information, see the `startapp` and `startproject` documentation.

Support for time zones

Django 1.4 adds *support for time zones*. When it's enabled, Django stores date and time information in UTC in the database, uses time zone-aware datetime objects internally, and translates them to the end user's time zone in templates and forms.

Reasons for using this feature include:

- Customizing date and time display for users around the world.
- Storing datetimes in UTC for database portability and interoperability. (This argument doesn't apply to PostgreSQL, because it already stores timestamps with time zone information in Django 1.3.)
- Avoiding data corruption problems around DST transitions.

Time zone support is enabled by default in new projects created with `startproject`. If you want to use this feature in an existing project, there is a *migration guide*.

Minor features

Django 1.4 also includes several smaller improvements worth noting:

- A more usable stacktrace in the technical 500 page: frames in the stack trace which reference Django's code are dimmed out, while frames in user code are slightly emphasized. This change makes it easier to scan a stacktrace for issues in user code.
- *Tablespace support* in PostgreSQL.
- Customizable names for `simple_tag()`.
- In the documentation, a helpful *security overview* page.
- The `django.contrib.auth.models.check_password()` function has been moved to the `django.contrib.auth.utils` module. Importing it from the old location will still work, but you should update your imports.
- The `collectstatic` management command gained a `--clear` option to delete all files at the destination before copying or linking the static files.
- It is now possible to load fixtures containing forward references when using MySQL with the InnoDB database engine.

- A new 403 response handler has been added as `'django.views.defaults.permission_denied'`. You can set your own handler by setting the value of `django.conf.urls.handler403`. See the documentation about *the 403 (HTTP Forbidden) view* for more information.
- The `trans` template tag now takes an optional `as` argument to be able to retrieve a translation string without displaying it but setting a template context variable instead.
- The `if` template tag now supports `{% elif %}` clauses.
- A new plain text version of the HTTP 500 status code internal error page served when `DEBUG` is `True` is now sent to the client when Django detects that the request has originated in JavaScript code (`is_ajax()` is used for this).

  Similarly to its HTML counterpart, it contains a collection of different pieces of information about the state of the web application.

  This should make it easier to read when debugging interaction with client-side Javascript code.
- Added the `--no-location` option to the `makemessages` command.
- Changed the `locmem` cache backend to use `pickle.HIGHEST_PROTOCOL` for better compatibility with the other cache backends.
- Added support in the ORM for generating `SELECT` queries containing `DISTINCT ON`.

  The `distinct()` `QuerySet` method now accepts an optional list of model field names. If specified, then the `DISTINCT` statement is limited to these fields. This is only supported in PostgreSQL.

  For more details, see the documentation for `distinct()`.

Backwards incompatible changes in 1.4

django.contrib.admin

The included administration app `django.contrib.admin` has for a long time shipped with a default set of static files such as JavaScript, images and stylesheets. Django 1.3 added a new contrib app `django.contrib.staticfiles` to handle such files in a generic way and defined conventions for static files included in apps.

Starting in Django 1.4 the admin's static files also follow this convention to make it easier to deploy the included files. In previous versions of Django, it was also common to define an `ADMIN_MEDIA_PREFIX` setting to point to the URL where the admin's static files are served by a web server. This setting has now been deprecated and replaced by the more general setting `STATIC_URL`. Django will now expect to find the admin static files under the URL `<STATIC_URL>/admin/`.

If you've previously used a URL path for `ADMIN_MEDIA_PREFIX` (e.g. `/media/`) simply make sure `STATIC_URL` and `STATIC_ROOT` are configured and your web server serves the files correctly. The development server continues to serve the admin files just like before. Don't hesitate to consult the *static files howto* for further details.

In case your `ADMIN_MEDIA_PREFIX` is set to an specific domain (e.g. `http://media.example.com/admin/`) make sure to also set your `STATIC_URL` setting to the correct URL, for example `http://media.example.com/`.

> **Warning:** If you're implicitly relying on the path of the admin static files on your server's file system when you deploy your site, you have to update that path. The files were moved from `django/contrib/admin/media/` to `django/contrib/admin/static/admin/`.

Supported browsers for the admin

Django hasn't had a clear policy on which browsers are supported for using the admin app. Django's new policy formalizes existing practices: YUI's A-grade browsers should provide a fully-functional admin experience, with the notable exception of IE6, which is no longer supported.

Released over ten years ago, IE6 imposes many limitations on modern web development. The practical implications of this policy are that contributors are free to improve the admin without consideration for these limitations.

This new policy **has no impact** on development outside of the admin. Users of Django are free to develop webapps compatible with any range of browsers.

Removed admin icons

As part of an effort to improve the performance and usability of the admin's changelist sorting interface and of the admin's `horizontal` and `vertical` "filter" widgets, some icon files were removed and grouped into two sprite files.

Specifically: `selector-add.gif`, `selector-addall.gif`, `selector-remove.gif`, `selector-removeall.gif`, `selector_stacked-add.gif` and `selector_stacked-remove.gif` were combined into `selector-icons.gif`; and `arrow-up.gif` and `arrow-down.gif` were combined into `sorting-icons.gif`.

If you used those icons to customize the admin then you will want to replace them with your own icons or retrieve them from a previous release.

CSS class names in admin forms

To avoid conflicts with other common CSS class names (e.g. "button"), a prefix "field-" has been added to all CSS class names automatically generated from the form field names in the main admin forms, stacked inline forms and tabular inline cells. You will need to take that prefix into account in your custom style sheets or javascript files if you previously used plain field names as selectors for custom styles or javascript transformations.

Compatibility with old signed data

Django 1.3 changed the cryptographic signing mechanisms used in a number of places in Django. While Django 1.3 kept fallbacks that would accept hashes produced by the previous methods, these fallbacks are removed in Django 1.4.

So, if you upgrade to Django 1.4 directly from 1.2 or earlier, you may lose/invalidate certain pieces of data that have been cryptographically signed using an old method. To avoid this, use Django 1.3 first for a period of time to allow the signed data to expire naturally. The affected parts are detailed below, with 1) the consequences of ignoring this advice and 2) the amount of time you need to run Django 1.3 for the data to expire or become irrelevant.

- `contrib.sessions` data integrity check
  - consequences: the user will be logged out, and session data will be lost.
  - time period: defined by `SESSION_COOKIE_AGE`.
- `contrib.auth` password reset hash
  - consequences: password reset links from before the upgrade will not work.
  - time period: defined by `PASSWORD_RESET_TIMEOUT_DAYS`.

Form-related hashes — these are much shorter lifetime, and are relevant only for the short window where a user might fill in a form generated by the pre-upgrade Django instance, and try to submit it to the upgraded Django instance:

- `contrib.comments` form security hash
  - consequences: the user will see a validation error "Security hash failed".
  - time period: the amount of time you expect users to take filling out comment forms.
- `FormWizard` security hash
  - consequences: the user will see an error about the form having expired, and will be sent back to the first page of the wizard, losing the data they have entered so far.
  - time period: the amount of time you expect users to take filling out the affected forms.
- CSRF check
  - Note: This is actually a Django 1.1 fallback, not Django 1.2, and applies only if you are upgrading from 1.1.
  - consequences: the user will see a 403 error with any CSRF protected POST form.
  - time period: the amount of time you expect user to take filling out such forms.

django.contrib.flatpages

Starting in the 1.4 release the `FlatpageFallbackMiddleware` only adds a trailing slash and redirects if the resulting URL refers to an existing flatpage. For example, requesting `/notaflatpageoravalidurl` in a previous version would redirect to `/notaflatpageoravalidurl/`, which would subsequently raise a 404. Requesting `/notaflatpageoravalidurl` now will immediately raise a 404. Additionally redirects returned by flatpages are now permanent (301 status code) to match the behavior of the `CommonMiddleware`.

Serialization of `datetime` and `time`

As a consequence of time zone support, and according to the ECMA-262 specification, some changes were made to the JSON serializer:

- It includes the time zone for aware datetime objects. It raises an exception for aware time objects.
- It includes milliseconds for datetime and time objects. There is still some precision loss, because Python stores microseconds (6 digits) and JSON only supports milliseconds (3 digits). However, it's better than discarding microseconds entirely.

The XML serializer was also changed to use the ISO8601 format for datetimes. The letter `T` is used to separate the date part from the time part, instead of a space. Time zone information is included in the `[+-]HH:MM` format.

The serializers will dump datetimes in fixtures with these new formats. They can still load fixtures that use the old format.

`supports_timezone` changed to `False` for SQLite

The database feature `supports_timezone` used to be `True` for SQLite. Indeed, if you saved an aware datetime object, SQLite stored a string that included an UTC offset. However, this offset was ignored when loading the value back from the database, which could corrupt the data.

In the context of time zone support, this flag was changed to `False`, and datetimes are now stored without time zone information in SQLite. When `USE_TZ` is `False`, if you attempt to save an aware datetime object, Django raises an exception.

Database connection's thread-locality

`DatabaseWrapper` objects (i.e. the connection objects referenced by `django.db.connection` and `django.db.connections["some_alias"]`) used to be thread-local. They are now global objects in order to be potentially shared between multiple threads. While the individual connection objects are now global, the `django.db.connections` dictionary referencing those objects is still thread-local. Therefore if you just use the ORM or `DatabaseWrapper.cursor()` then the behavior is still the same as before. Note, however, that `django.db.connection` does not directly reference the default `DatabaseWrapper` object anymore and is now a proxy to access that object's attributes. If you need to access the actual `DatabaseWrapper` object, use `django.db.connections[DEFAULT_DB_ALIAS]` instead.

As part of this change, all underlying SQLite connections are now enabled for potential thread-sharing (by passing the `check_same_thread=False` attribute to pysqlite). `DatabaseWrapper` however preserves the previous behavior by disabling thread-sharing by default, so this does not affect any existing code that purely relies on the ORM or on `DatabaseWrapper.cursor()`.

Finally, while it is now possible to pass connections between threads, Django does not make any effort to synchronize access to the underlying backend. Concurrency behavior is defined by the underlying backend implementation. Check their documentation for details.

COMMENTS_BANNED_USERS_GROUP setting

Django's *comments app* has historically supported excluding the comments of a special user group, but we've never documented the feature properly and didn't enforce the exclusion in other parts of the app such as the template tags. To fix this problem, we removed the code from the feed class.

If you rely on the feature and want to restore the old behavior, simply use a custom comment model manager to exclude the user group, like this:

```
from django.conf import settings
from django.contrib.comments.managers import CommentManager

class BanningCommentManager(CommentManager):
    def get_query_set(self):
        qs = super(BanningCommentManager, self).get_query_set()
        if getattr(settings, 'COMMENTS_BANNED_USERS_GROUP', None):
            where = ['user_id NOT IN (SELECT user_id FROM auth_user_groups WHERE group_id = %s)']
            params = [settings.COMMENTS_BANNED_USERS_GROUP]
            qs = qs.extra(where=where, params=params)
        return qs
```

Save this model manager in your custom comment app (e.g. in `my_comments_app/managers.py`) and add it your *custom comment app model*:

```
from django.db import models
from django.contrib.comments.models import Comment

from my_comments_app.managers import BanningCommentManager

class CommentWithTitle(Comment):
    title = models.CharField(max_length=300)

    objects = BanningCommentManager()
```

For more details, see the documentation about *customizing the comments framework*.

IGNORABLE_404_STARTS and *IGNORABLE_404_ENDS* settings

Until Django 1.3, it was possible to exclude some URLs from Django's *404 error reporting* by adding prefixes to `IGNORABLE_404_STARTS` and suffixes to `IGNORABLE_404_ENDS`.

In Django 1.4, these two settings are superseded by `IGNORABLE_404_URLS`, which is a list of compiled regular expressions. Django won't send an email for 404 errors on URLs that match any of them.

Furthermore, the previous settings had some rather arbitrary default values:

```
IGNORABLE_404_STARTS = ('/cgi-bin/', '/_vti_bin', '/_vti_inf')
IGNORABLE_404_ENDS = ('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi',
                      'favicon.ico', '.php')
```

It's not Django's role to decide if your website has a legacy `/cgi-bin/` section or a `favicon.ico`. As a consequence, the default values of `IGNORABLE_404_URLS`, `IGNORABLE_404_STARTS` and `IGNORABLE_404_ENDS` are all now empty.

If you have customized `IGNORABLE_404_STARTS` or `IGNORABLE_404_ENDS`, or if you want to keep the old default value, you should add the following lines in your settings file:

```
import re
IGNORABLE_404_URLS = (
    # for each <prefix> in IGNORABLE_404_STARTS
    re.compile(r'^<prefix>'),
    # for each <suffix> in IGNORABLE_404_ENDS
    re.compile(r'<suffix>$'),
)
```

Don't forget to escape characters that have a special meaning in a regular expression.

CSRF protection extended to PUT and DELETE

Previously, Django's *CSRF protection* provided protection against only POST requests. Since use of PUT and DELETE methods in AJAX applications is becoming more common, we now protect all methods not defined as safe by **RFC 2616** i.e. we exempt GET, HEAD, OPTIONS and TRACE, and enforce protection on everything else.

If you are using PUT or DELETE methods in AJAX applications, please see the *instructions about using AJAX and CSRF*.

`django.core.template_loaders`

This was an alias to `django.template.loader` since 2005, it has been removed without emitting a warning due to the length of the deprecation. If your code still referenced this please use `django.template.loader` instead.

`django.db.models.fields.URLField.verify_exists`

This functionality has been removed due to intractable performance and security issues. Any existing usage of `verify_exists` should be removed.

`django.core.files.storage.Storage.open`

The `open` method of the base Storage class took an obscure parameter `mixin` which allowed you to dynamically change the base classes of the returned file object. This has been removed. In the rare case you relied on the *mixin*

parameter, you can easily achieve the same by overriding the *open* method, e.g.:

```
from django.core.files import File
from django.core.files.storage import FileSystemStorage

class Spam(File):
    """
    Spam, spam, spam, spam and spam.
    """
    def ham(self):
        return 'eggs'

class SpamStorage(FileSystemStorage):
    """
    A custom file storage backend.
    """
    def open(self, name, mode='rb'):
        return Spam(open(self.path(name), mode))
```

YAML deserializer now uses `yaml.safe_load`

`yaml.load` is able to construct any Python object, which may trigger arbitrary code execution if you process a YAML document that comes from an untrusted source. This feature isn't necessary for Django's YAML deserializer, whose primary use is to load fixtures consisting of simple objects. Even though fixtures are trusted data, for additional security, the YAML deserializer now uses `yaml.safe_load`.

Features deprecated in 1.4

Old styles of calling `cache_page` decorator

Some legacy ways of calling `cache_page()` have been deprecated, please see the docs for the correct way to use this decorator.

Support for PostgreSQL versions older than 8.2

Django 1.3 dropped support for PostgreSQL versions older than 8.0 and the relevant documents suggested to use a recent version because of performance reasons but more importantly because end of the upstream support periods for releases 8.0 and 8.1 was near (November 2010).

Django 1.4 takes that policy further and sets 8.2 as the minimum PostgreSQL version it officially supports.

Request exceptions are now always logged

When *logging support* was added to Django in 1.3, the admin error email support was moved into the `django.utils.log.AdminEmailHandler`, attached to the `'django.request'` logger. In order to maintain the established behavior of error emails, the `'django.request'` logger was called only when `DEBUG` was `False`.

To increase the flexibility of error logging for requests, the `'django.request'` logger is now called regardless of the value of `DEBUG`, and the default settings file for new projects now includes a separate filter attached to `django.utils.log.AdminEmailHandler` to prevent admin error emails in `DEBUG` mode:

```
'filters': {
    'require_debug_false': {
        '()': 'django.utils.log.RequireDebugFalse'
    }
},
'handlers': {
    'mail_admins': {
        'level': 'ERROR',
        'filters': ['require_debug_false'],
        'class': 'django.utils.log.AdminEmailHandler'
    }
},
```

If your project was created prior to this change, your `LOGGING` setting will not include this new filter. In order to maintain backwards-compatibility, Django will detect that your `'mail_admins'` handler configuration includes no `'filters'` section, and will automatically add this filter for you and issue a pending-deprecation warning. This will become a deprecation warning in Django 1.5, and in Django 1.6 the backwards-compatibility shim will be removed entirely.

The existence of any `'filters'` key under the `'mail_admins'` handler will disable this backward-compatibility shim and deprecation warning.

`django.conf.urls.defaults`

Until Django 1.3 the functions `include()`, `patterns()` and `url()` plus `handler404`, `handler500` were located in a `django.conf.urls.defaults` module.

Starting with Django 1.4 they are now available in `django.conf.urls`.

`django.contrib.databrowse`

Databrowse has not seen active development for some time, and this does not show any sign of changing. There had been a suggestion for a GSOC project to integrate the functionality of databrowse into the admin, but no progress was made. While Databrowse has been deprecated, an enhancement of `django.contrib.admin` providing a similar feature set is still possible.

The code that powers Databrowse is licensed under the same terms as Django itself, and so is available to be adopted by an individual or group as a third-party project.

`django.core.management.setup_environ`

This function temporarily modified `sys.path` in order to make the parent "project" directory importable under the old flat `startproject` layout. This function is now deprecated, as its path workarounds are no longer needed with the new `manage.py` and default project layout.

This function was never documented or part of the public API, but was widely recommended for use in setting up a "Django environment" for a user script. These uses should be replaced by setting the `DJANGO_SETTINGS_MODULE` environment variable or using `django.conf.settings.configure()`.

`django.core.management.execute_manager`

This function was previously used by `manage.py` to execute a management command. It is identical to `django.core.management.execute_from_command_line`, except that it first calls `setup_environ`,

which is now deprecated. As such, `execute_manager` is also deprecated; `execute_from_command_line` can be used instead. Neither of these functions is documented as part of the public API, but a deprecation path is needed due to use in existing `manage.py` files.

`is_safe` and `needs_autoescape` attributes of template filters

Two flags, `is_safe` and `needs_autoescape`, define how each template filter interacts with Django's auto-escaping behavior. They used to be attributes of the filter function:

```
@register.filter
def noop(value):
    return value
noop.is_safe = True
```

However, this technique caused some problems in combination with decorators, especially `@stringfilter`. Now, the flags are keyword arguments of `@register.filter`:

```
@register.filter(is_safe=True)
def noop(value):
    return value
```

See *filters and auto-escaping* for more information.

Session cookies now have the `httponly` flag by default

Session cookies now include the `httponly` attribute by default to help reduce the impact of potential XSS attacks. For strict backwards compatibility, use `SESSION_COOKIE_HTTPONLY = False` in your settings file.

Wildcard expansion of application names in *INSTALLED_APPS*

Until Django 1.3, `INSTALLED_APPS` accepted wildcards in application names, like `django.contrib.*`. The expansion was performed by a filesystem-based implementation of `from <package> import *`. Unfortunately, this can't be done reliably.

This behavior was never documented. Since it is un-pythonic and not obviously useful, it was removed in Django 1.4. If you relied on it, you must edit your settings file to list all your applications explicitly.

`HttpRequest.raw_post_data` renamed to `HttpRequest.body`

This attribute was confusingly named `HttpRequest.raw_post_data`, but it actually provided the body of the HTTP request. It's been renamed to `HttpRequest.body`, and `HttpRequest.raw_post_data` has been deprecated.

The Django 1.4 roadmap

Before the final Django 1.4 release, several other preview/development releases will be made available. The current schedule consists of at least the following:

- Week of **January 30, 2012**: First Django 1.4 beta release; final feature freeze for Django 1.4.
- Week of **February 27, 2012**: First Django 1.4 release candidate; string freeze for translations.
- Week of **March 5, 2012**: Django 1.4 final release.

If necessary, additional alpha, beta or release-candidate packages will be issued prior to the final 1.4 release. Django 1.4 will be released approximately one week after the final release candidate.

What you can do to help

In order to provide a high-quality 1.4 release, we need your help. Although this alpha release is, again, *not* intended for production use, you can help the Django team by trying out the alpha codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.3 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Several development sprints will also be taking place before the 1.4 release; these will typically be announced in advance on the django-developers mailing list, and anyone who wants to help is welcome to join in.

9.2.3 Django 1.3 beta 1 release notes

Welcome to Django 1.3 beta 1!

This is the second in a series of preview/development releases leading up to the eventual release of Django 1.3. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.3 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

What's new in Django 1.3 beta 1

Further tweaks to the staticfiles app

Django 1.3 ships with a new contrib app `django.contrib.staticfiles` to help developers handle the static media files (images, CSS, JavaScript, etc.) that are needed to render a complete web page.

The `staticfiles` app ships with the ability to automatically serve static files during development (if the `DEBUG` setting is `True`) when using the `runserver` management command. Based on feedback from the community this release adds two new options to the `runserver` command to modify this behavior:

- `--nostatic`: prevents the `runserver` command from serving files completely.
- `--insecure`: enables serving of static files even if running with `DEBUG` set to False. (This is **not** recommended!)

See the *staticfiles reference documentation* for more details, or learn *how to manage static files*.

Translation comments

If you would like to give translators hints about a translatable string, you can add a comment prefixed with the `Translators` keyword on the line preceding the string, e.g.:

```
def my_view(request):
    # Translators: This message appears on the home page only
    output = ugettext("Welcome to my site.")
```

The comment will appear in the resulting .po file and should also be displayed by most translation tools.

For more information, see *Comments for translators*.

Permissions for inactive users

If you provide a custom auth backend with `supports_inactive_user` set to `True`, an inactive user model will check the backend for permissions. This is useful for further centralizing the permission handling. See the *authentication docs* for more details.

Backwards-incompatible changes in 1.3 alpha 2

Change to admin lookup filters

The Django admin has long had an undocumented "feature" allowing savvy users to manipulate the query string of changelist pages to filter the list of objects displayed. However, this also creates a security issue, as a staff user with sufficient knowledge of model structure could use this "feature" to gain access to information he or she would not normally have.

As a result, changelist filtering now explicitly validates all lookup arguments in the query string, and permits only fields which are directly on the model, or relations explicitly permitted by the `ModelAdmin` definition. If you were relying on this undocumented feature, you will need to update your `ModelAdmin` definitions to whitelist the relations you choose to expose for filtering.

Introduction of STATIC_URL and STATIC_ROOT settings

The newly introduced `staticfiles` app – which extends Django's abilities to handle static files for apps and projects – required the additon of two new settings to refer to those files in templates and code, especially in contrast to the `MEDIA_URL` and `MEDIA_ROOT` settings that refer to user-uploaded files.

Prior to 1.3 alpha 2 these settings were called `STATICFILES_URL` and `STATICFILES_ROOT` to follow the naming scheme for app-centric settings. Based on feedback from the community it became apparent that those settings created confusion, especially given the fact that handling static files is also desired outside the use of the optional `staticfiles` app.

As a result, we took the following steps to rectify the issue:

- Two new global settings were added that will be used by, **but are not limited to**, the *staticfiles* app:
- `STATIC_ROOT` (formally `STATICFILES_ROOT`)
- `STATIC_URL` (formally `STATICFILES_URL`)
- The `django.contrib.staticfiles.templatetags.staticfiles.get_staticfiles_prefix` template tag was moved to Django's core (`django.templatetags.static`) and renamed to `get_static_prefix`.

- The `django.contrib.staticfiles.context_processors.staticfiles` context processor was moved to Django's core (`django.core.context_processors.static`) and renamed to `static()`.
- *Paths in media definitions* now uses `STATIC_URL` as the prefix **if the value is not None**, and falls back to the previously used `MEDIA_URL` setting otherwise.

Changes to the login methods of the admin

In previous version the admin app defined login methods in multiple locations and ignored the almost identical implementation in the already used auth app. A side effect of this duplication was the missing adoption of the changes made in r12634 to support a broader set of characters for usernames.

This release refactors the admin's login mechanism to use a subclass of the `AuthenticationForm` instead of a manual form validation. The previously undocumented method `'django.contrib.admin.sites.AdminSite.display_login_form'` has been removed in favor of a new `login_form` attribute.

Changes to `USStateField`

The `django.contrib.localflavor` application contains collections of code relevant to specific countries or cultures. One such is `USStateField`, which provides a field for storing the two-letter postal abbreviation of a U.S. state. This field has consistently caused problems, however, because it is often used to store the state portion of a U.S postal address, but not all "states" recognized by the U.S Postal Service are actually states of the U.S. or even U.S. territory. Several compromises over the list of choices resulted in some users feeling the field supported too many locations, while others felt it supported too few.

In Django 1.3 we're taking a new approach to this problem, implemented as a pair of changes:

- The choice list for *USStateField* has changed. Previously, it consisted of the 50 U.S. states, the District of Columbia and U.S. overseas territories. As of Django 1.3 it includes all previous choices, plus the U.S. Armed Forces postal codes.
- A new model field, `django.contrib.localflavor.us.models.USPostalCodeField`, has been added which draws its choices from a list of all postal abbreviations recognized by the U.S Postal Service. This includes all abbreviations recognized by *USStateField*, plus three independent nations – the Federated States of Micronesia, the Republic of the Marshall Islands and the Republic of Palau – which are serviced under treaty by the U.S. postal system. A new form widget, `django.contrib.localflavor.us.forms.USPSSelect`, is also available and provides the same set of choices.

Additionally, several finer-grained choice tuples are provided which allow mixing and matching of subsets of the U.S. states and territories, and other locations serviced by the U.S. postal system. Consult the `django.contrib.localflavor` documentation for more details.

The change to *USStateField* is technically backwards-incompatible for users who expect this field to exclude Armed Forces locations. If you need to support U.S. mailing addresses without Armed Forces locations, see the list of choice tuples available in the localflavor documentation.

The Django 1.3 roadmap

Before the final Django 1.3 release, several other preview/development releases will be made available. The current schedule consists of at least the following:

- Week of **January 24, 2011**: First Django 1.3 release candidate. String freeze for translations.

- Week of **January 31, 2011**: Django 1.3 final release.

If necessary, additional beta or release-candidate packages will be issued prior to the final 1.3 release. Django 1.3 will be released approximately one week after the final release candidate.

What you can do to help

In order to provide a high-quality 1.3 release, we need your help. Although this beta release is, again, *not* intended for production use, you can help the Django team by trying out the beta codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.3 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

9.2.4 Django 1.3 alpha 1 release notes

November 11, 2010

Welcome to Django 1.3 alpha 1!

This is the first in a series of preview/development releases leading up to the eventual release of Django 1.3. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.3 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

As of this alpha release, Django 1.3 contains a number of nifty new features, lots of bug fixes, some minor backwards incompatible changes and an easy upgrade path from Django 1.2.

What's new in Django 1.3 alpha 1

Class-based views

Django 1.3 adds a framework that allows you to use a class as a view. This means you can compose a view out of a collection of methods that can be subclassed and overridden to provide common views of data without having to write too much code.

Analogs of all the old function-based generic views have been provided, along with a completely generic view base class that can be used as the basis for reusable applications that can be easily extended.

See *the documentation on Class-based Generic Views* for more details. There is also a document to help you convert your function-based generic views to class-based views.

Logging

Django 1.3 adds framework-level support for Python's logging module. This means you can now easily configure and control logging as part of your Django project. A number of logging handlers and logging calls have been added to Django's own code as well – most notably, the error emails sent on a HTTP 500 server error are now handled as a logging activity. See *the documentation on Django's logging interface* for more details.

Extended static files handling

Django 1.3 ships with a new contrib app `'django.contrib.staticfiles'` to help developers handle the static media files (images, CSS, Javascript, etc.) that are needed to render a complete web page.

In previous versions of Django, it was common to place static assets in `MEDIA_ROOT` along with user-uploaded files, and serve them both at `MEDIA_URL`. Part of the purpose of introducing the `staticfiles` app is to make it easier to keep static files separate from user-uploaded files. For this reason, you will probably want to make your `MEDIA_ROOT` and `MEDIA_URL` different from your `STATICFILES_ROOT` and `STATICFILES_URL`. You will need to arrange for serving of files in `MEDIA_ROOT` yourself; `staticfiles` does not deal with user-uploaded media at all.

See the *reference documentation of the app* for more details or learn how to *manage static files*.

`unittest2` support

Python 2.7 introduced some major changes to the unittest library, adding some extremely useful features. To ensure that every Django project can benefit from these new features, Django ships with a copy of unittest2, a copy of the Python 2.7 unittest library, backported for Python 2.4 compatibility.

To access this library, Django provides the `django.utils.unittest` module alias. If you are using Python 2.7, or you have installed unittest2 locally, Django will map the alias to the installed version of the unittest library. Otherwise, Django will use it's own bundled version of unittest2.

To use this alias, simply use:

```
from django.utils import unittest
```

wherever you would have historically used:

```
import unittest
```

If you want to continue to use the base unittest libary, you can – you just won't get any of the nice new unittest2 features.

Transaction context managers

Users of Python 2.5 and above may now use *transaction management functions* as context managers. For example:

```
with transaction.autocommit():
    # ...
```

For more information, see *Controlling transaction management in views*.

Configurable delete-cascade

`ForeignKey` and `OneToOneField` now accept an `on_delete` argument to customize behavior when the referenced object is deleted. Previously, deletes were always cascaded; available alternatives now include set null, set

default, set to any value, protect, or do nothing.

For more information, see the `on_delete` documentation.

Contextual markers in translatable strings

For translation strings with ambiguous meaning, you can now use the `pgettext` function to specify the context of the string.

For more information, see *Contextual markers*

Everything else

Django *1.1* and *1.2* added lots of big ticket items to Django, like multiple-database support, model validation, and a session-based messages framework. However, this focus on big features came at the cost of lots of smaller features.

To compensate for this, the focus of the Django 1.3 development process has been on adding lots of smaller, long standing feature requests. These include:

- Improved tools for accessing and manipulating the current Site via `django.contrib.sites.models.get_current_site()`.
- A `RequestFactory` for mocking requests in tests.
- A new test assertion – `assertNumQueries()` – making it easier to test the database activity associated with a view.

Backwards-incompatible changes in 1.3 alpha 1

PasswordInput default rendering behavior

The `PasswordInput` form widget, intended for use with form fields which represent passwords, accepts a boolean keyword argument `render_value` indicating whether to send its data back to the browser when displaying a submitted form with errors. Prior to Django 1.3, this argument defaulted to `True`, meaning that the submitted password would be sent back to the browser as part of the form. Developers who wished to add a bit of additional security by excluding that value from the redisplayed form could instantiate a `PasswordInput` passing `render_value=False`.

Due to the sensitive nature of passwords, however, Django 1.3 takes this step automatically; the default value of `render_value` is now `False`, and developers who want the password value returned to the browser on a submission with errors (the previous behavior) must now explicitly indicate this. For example:

```
class LoginForm(forms.Form):
    username = forms.CharField(max_length=100)
    password = forms.CharField(widget=forms.PasswordInput(render_value=True))
```

Clearable default widget for FileField

Django 1.3 now includes a `ClearableFileInput` form widget in addition to `FileInput`. `ClearableFileInput` renders with a checkbox to clear the field's value (if the field has a value and is not required); `FileInput` provided no means for clearing an existing file from a `FileField`.

`ClearableFileInput` is now the default widget for a `FileField`, so existing forms including `FileField` without assigning a custom widget will need to account for the possible extra checkbox in the rendered form output.

To return to the previous rendering (without the ability to clear the `FileField`), use the `FileInput` widget in place of `ClearableFileInput`. For instance, in a `ModelForm` for a hypothetical `Document` model with a `FileField` named `document`:

```
from django import forms
from myapp.models import Document

class DocumentForm(forms.ModelForm):
    class Meta:
        model = Document
        widgets = {'document': forms.FileInput}
```

New index on database session table

Prior to Django 1.3, the database table used by the database backend for the *sessions* app had no index on the `expire_date` column. As a result, date-based queries on the session table – such as the query that is needed to purge old sessions – would be very slow if there were lots of sessions.

If you have an existing project that is using the database session backend, you don't have to do anything to accommodate this change. However, you may get a significant performance boost if you manually add the new index to the session table. The SQL that will add the index can be found by running the `sqlindexes` admin command:

```
python manage.py sqlindexes sessions
```

No more naughty words

Django has historically provided (and enforced) a list of profanities. The *comments app* has enforced this list of profanities, preventing people from submitting comments that contained one of those profanities.

Unfortunately, the technique used to implement this profanities list was woefully naive, and prone to the Scunthorpe problem. Fixing the built in filter to fix this problem would require significant effort, and since natural language processing isn't the normal domain of a web framework, we have "fixed" the problem by making the list of prohibited words an empty list.

If you want to restore the old behavior, simply put a `PROFANITIES_LIST` setting in your settings file that includes the words that you want to prohibit (see the commit that implemented this change if you want to see the list of words that was historically prohibited). However, if avoiding profanities is important to you, you would be well advised to seek out a better, less naive approach to the problem.

Localflavor changes

Django 1.3 introduces the following backwards-incompatible changes to local flavors:

- Indonesia (id) – The province "Nanggroe Aceh Darussalam (NAD)" has been removed from the province list in favor of the new official designation "Aceh (ACE)".

Features deprecated in 1.3

Django 1.3 deprecates some features from earlier releases. These features are still supported, but will be gradually phased out over the next few release cycles.

Code taking advantage of any of the features below will raise a `PendingDeprecationWarning` in Django 1.3. This warning will be silent by default, but may be turned on using Python's `warnings` module, or by running Python with a `-Wd` or *-Wall* flag.

In Django 1.4, these warnings will become a `DeprecationWarning`, which is *not* silent. In Django 1.5 support for these features will be removed entirely.

See Also:

For more details, see the documentation *Django's release process* and our *deprecation timeline*.

`mod_python` support

The `mod_python` library has not had a release since 2007 or a commit since 2008. The Apache Foundation board voted to remove `mod_python` from the set of active projects in its version control repositories, and its lead developer has shifted all of his efforts toward the lighter, slimmer, more stable, and more flexible `mod_wsgi` backend.

If you are currently using the `mod_python` request handler, you are strongly encouraged to redeploy your Django instances using *mod_wsgi*.

Function-based generic views

As a result of the introduction of class-based generic views, the function-based generic views provided by Django have been deprecated. The following modules and the views they contain have been deprecated:

- `django.views.generic.create_update`
- `django.views.generic.date_based`
- `django.views.generic.list_detail`
- `django.views.generic.simple`

Test client response `template` attribute

Django's *test client* returns `Response` objects annotated with extra testing information. In Django versions prior to 1.3, this included a `template` attribute containing information about templates rendered in generating the response: either None, a single `Template` object, or a list of `Template` objects. This inconsistency in return values (sometimes a list, sometimes not) made the attribute difficult to work with.

In Django 1.3 the `template` attribute is deprecated in favor of a new `templates` attribute, which is always a list, even if it has only a single element or no elements.

`DjangoTestRunner`

As a result of the introduction of support for unittest2, the features of `django.test.simple.DjangoTestRunner` (including fail-fast and Ctrl-C test termination) have been made redundant. In view of this redundancy, `DjangoTestRunner` has been turned into an empty placeholder class, and will be removed entirely in Django 1.5.

The Django 1.3 roadmap

Before the final Django 1.3 release, several other preview/development releases will be made available. The current schedule consists of at least the following:

- Week of **November 29, 2010**: First Django 1.3 beta release. Final feature freeze for Django 1.3.
- Week of **January 10, 2011**: First Django 1.3 release candidate. String freeze for translations.

- Week of **January 17, 2011**: Django 1.3 final release.

If necessary, additional alpha, beta or release-candidate packages will be issued prior to the final 1.3 release. Django 1.3 will be released approximately one week after the final release candidate.

What you can do to help

In order to provide a high-quality 1.3 release, we need your help. Although this alpha release is, again, *not* intended for production use, you can help the Django team by trying out the alpha codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.3 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Several development sprints will also be taking place before the 1.3 release; these will typically be announced in advance on the django-developers mailing list, and anyone who wants to help is welcome to join in.

9.2.5 Django 1.2 RC 1 release notes

May 5, 2010

Welcome to the first Django 1.2 release candidate!

This is the third – and likely last – in a series of preview/development releases leading up to the eventual release of Django 1.2. This release is targeted primarily at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve any critical bugs prior to the final 1.2 release.

As such, this release is not yet intended for production use, and any such use is discouraged.

Django has been feature frozen since the 1.2 beta release, so this release candidate contains no new features, only bugfixes; for a summary of features new to Django 1.2, consult the *1.2 alpha* and *1.2 beta* release notes.

Python compatibility

While not a new feature, it's important to note that Django 1.2 introduces the first shift in our Python compatibility policy since Django's initial public debut. Previous Django releases were tested and supported on 2.x Python versions from 2.3 up; Django 1.2, however, drops official support for Python 2.3. As such, the minimum Python version required for Django is now 2.4, and Django is tested and supported on Python 2.4, 2.5 and 2.6, and will be supported on the as-yet-unreleased Python 2.7.

This change should affect only a small number of Django users, as most operating-system vendors today are shipping Python 2.4 or newer as their default version. If you're still using Python 2.3, however, you'll need to stick to Django

1.1 until you can upgrade; per *our support policy*, Django 1.1 will continue to receive security support until the release of Django 1.3.

A roadmap for Django's overall 2.x Python support, and eventual transition to Python 3.x, is currently being developed, and will be announced prior to the release of Django 1.3.

The Django 1.2 roadmap

As of this release candidate, Django 1.2 is in both feature freeze and "string freeze" – all strings marked for translation in the Django codebase will retain their current form in the final Django 1.2 release. Only critical release-blocking bugs, documentation and updated translation files will receive attention between now and the final 1.2 release. Note that Django's localization infrastructure has been expanded for 1.2, and translation packages should now include a `formats.py` file containing data for localized formatting of numbers and dates.

If no critical bugs are discovered, Django 1.2 will be released approximately one week after this release candidate, on or about May 12, 2010.

What you can do to help

In order to provide a high-quality 1.2 release, we need your help. Although this release candidate is, again, *not* intended for production use, you can help the Django team by trying out this release candidate in a safe testing environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open a new ticket only if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.2 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

9.2.6 Django 1.2 beta 1 release notes

February 5, 2010

Welcome to Django 1.2 beta 1!

This is the second in a series of preview/development releases leading up to the eventual release of Django 1.2, currently scheduled to take place in March 2010. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.2 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

This document covers changes since the Django 1.2 alpha release; the *1.2 alpha release notes* cover new and updated features in Django between 1.1 and 1.2 alpha.

Deprecations and other changes in 1.2 beta

This beta release deprecates two portions of public API, and introduces a potentially backwards-incompatible change to another. Under *our API stability policy*, deprecation proceeds over multiple release cycles: initially, the deprecated API will raise `PendingDeprecationWarning`, followed by raising `DeprecationWarning` in the next release, and finally removal of the deprecated API in the release after that. APIs beginning the deprecation process in Django 1.2 will be removed in the Django 1.4 release.

Unit test runners

Django 1.2 changes the test runner tools to use a class-based approach. Old style function-based test runners will still work, but should be updated to use the new *class-based runners*.

Syndication feeds

The `django.contrib.syndication.feeds.Feed` class is being replaced by the `django.contrib.syndication.views.Feed` class. The old `feeds.Feed` class is deprecated. The new class has an almost identical API, but allows instances to be used as views.

Also, in accordance with RSS best practices, RSS feeds will now include an `atom:link` element. You may need to update your tests to take this into account.

For more information, see the full *syndication framework documentation*.

Cookie encoding

Due to cookie-handling bugs in Internet Explorer, Safari, and possibly other browsers, Django's encoding of cookie values was changed so that the characters comma (',') and semi-colon (';') are treated as non-safe characters, and are therefore encoded as `\054` and `\073` respectively. This could produce backwards incompatibilities if you are relying on the ability to set these characters directly in cookie values.

What's new in 1.2 beta

This 1.2 beta release marks the final feature freeze for Django 1.2; while most feature development was completed for 1.2 alpha (which constituted a freeze on major features), a few other new features were added afterward and so are new as of 1.2 beta.

Object-level permissions

A foundation for specifying permissions at the per-object level was added in Django 1.2 alpha but not documented with the alpha release.

The default authentication backends shipped with Django do not currently make use of this, but third-party authentication backends are free to do so. See the *authentication docs* for more information.

Permissions for anonymous users

If you provide a custom authentication backend with the attribute `supports_anonymous_user` set to `True`, the `AnonymousUser` class will check the backend for permissions, just as the normal `User` does. This is intended to

help centralize permission handling; apps can always delegate the question of whether something is allowed or not to the authorization/authentication system. See the *authentication docs* for more details.

`select_related()` improvements

The `select_related()` method of `QuerySet` now accepts the `related_name` of a reverse one-to-one relation in the list of fields to select. One-to-one relations will not, however, be traversed by a depth-based `select_related()` call.

The Django 1.2 roadmap

Before the final Django 1.2 release, at least one additional preview/development releases will be made available. The current schedule consists of at least the following:

- Week of **March 2, 2010**: First Django 1.2 release candidate. String freeze for translations.
- Week of **March 9, 2010**: Django 1.2 final release.

If necessary, additional beta or release-candidate packages will be issued prior to the final 1.2 release. Django 1.2 will be released approximately one week after the final release candidate.

What you can do to help

In order to provide a high-quality 1.2 release, we need your help. Although this beta release is, again, *not* intended for production use, you can help the Django team by trying out the beta codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.2 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Development sprints for Django 1.2 will also be taking place at PyCon US 2010, on the dedicated sprint days (February 22 through 25), and anyone who wants to help out is welcome to join in, either in person at PyCon or virtually in the IRC channel or on the mailing list.

9.2.7 Django 1.2 alpha 1 release notes

January 5, 2010

Welcome to Django 1.2 alpha 1!

This is the first in a series of preview/development releases leading up to the eventual release of Django 1.2, currently scheduled to take place in March 2010. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.2 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

Backwards-incompatible changes in 1.2

CSRF Protection

There have been large changes to the way that CSRF protection works, detailed in *the CSRF documentation*. The following are the major changes that developers must be aware of:

- `CsrfResponseMiddleware` and `CsrfMiddleware` have been deprecated, and **will be removed completely in Django 1.4**, in favor of a template tag that should be inserted into forms.
- All contrib apps use a `csrf_protect` decorator to protect the view. This requires the use of the `csrf_token` template tag in the template, so if you have used custom templates for contrib views, you MUST READ THE UPGRADE INSTRUCTIONS to fix those templates.

  Documentation removed

  The upgrade notes have been removed in current Django docs. Please refer to the docs for Django 1.3 or older to find these instructions.

- `CsrfViewMiddleware` is included in `MIDDLEWARE_CLASSES` by default. This turns on CSRF protection by default, so that views that accept POST requests need to be written to work with the middleware. Instructions on how to do this are found in the CSRF docs.
- CSRF-related code has moved from `contrib` to `core` (with backwards compatible imports in the old locations, which are deprecated).

`if` tag changes

Due to new features in the `if` template tag, it no longer accepts 'and', 'or' and 'not' as valid **variable** names. Previously that worked in some cases even though these strings were normally treated as keywords. Now, the keyword status is always enforced, and template code like `{% if not %}` or `{% if and %}` will throw a TemplateSyntaxError.

`LazyObject`

`LazyObject` is an undocumented utility class used for lazily wrapping other objects of unknown type. In Django 1.1 and earlier, it handled introspection in a non-standard way, depending on wrapped objects implementing a public method `get_all_members()`. Since this could easily lead to name clashes, it has been changed to use the standard method, involving `__members__` and `__dir__()`. If you used `LazyObject` in your own code, and implemented the `get_all_members()` method for wrapped objects, you need to make the following changes:

- If your class does not have special requirements for introspection (i.e. you have not implemented `__getattr__()` or other methods that allow for attributes not discoverable by normal mechanisms), you can simply remove the `get_all_members()` method. The default implementation on `LazyObject` will do the right thing.
- If you have more complex requirements for introspection, first rename the `get_all_members()` method to `__dir__()`. This is the standard method, from Python 2.6 onwards, for supporting introspection. If you are require support for Python < 2.6, add the following code to the class:

```
    __members__ = property(lambda self: self.__dir__())
```

`__dict__` on Model instances

Historically, the `__dict__` attribute of a model instance has only contained attributes corresponding to the fields on a model.

In order to support multiple database configurations, Django 1.2 has added a `_state` attribute to object instances. This attribute will appear in `__dict__` for a model instance. If your code relies on iterating over __dict__ to obtain a list of fields, you must now filter the `_state` attribute of out `__dict__`.

`get_db_prep_*()` methods on Field

Prior to v1.2, a custom field had the option of defining several functions to support conversion of Python values into database-compatible values. A custom field might look something like:

```
class CustomModelField(models.Field):
    # ...

    def get_db_prep_save(self, value):
        # ...

    def get_db_prep_value(self, value):
        # ...

    def get_db_prep_lookup(self, lookup_type, value):
        # ...
```

In 1.2, these three methods have undergone a change in prototype, and two extra methods have been introduced:

```
class CustomModelField(models.Field):
    # ...

    def get_prep_value(self, value):
        # ...

    def get_prep_lookup(self, lookup_type, value):
        # ...

    def get_db_prep_save(self, value, connection):
        # ...

    def get_db_prep_value(self, value, connection, prepared=False):
        # ...

    def get_db_prep_lookup(self, lookup_type, value, connection, prepared=False):
        # ...
```

These changes are required to support multiple databases: `get_db_prep_*` can no longer make any assumptions regarding the database for which it is preparing. The `connection` argument now provides the preparation methods with the specific connection for which the value is being prepared.

The two new methods exist to differentiate general data preparation requirements, and requirements that are database-specific. The `prepared` argument is used to indicate to the database preparation methods whether generic value preparation has been performed. If an unprepared (i.e., `prepared=False`) value is provided to the

`get_db_prep_*()` calls, they should invoke the corresponding `get_prep_*()` calls to perform generic data preparation.

Conversion functions has been provided which will transparently convert functions adhering to the old prototype into functions compatible with the new prototype. However, this conversion function will be removed in Django 1.4, so you should upgrade your Field definitions to use the new prototype.

If your `get_db_prep_*()` methods made no use of the database connection, you should be able to upgrade by renaming `get_db_prep_value()` to `get_prep_value()` and `get_db_prep_lookup()` to `get_prep_lookup()`. If you require database specific conversions, then you will need to provide an implementation ``get_db_prep_*` that uses the `connection` argument to resolve database-specific values.

Stateful template tags

Template tags that store rendering state on the node itself may experience problems if they are used with the new *cached template loader*.

All of the built-in Django template tags are safe to use with the cached loader, but if you're using custom template tags that come from third party packages, or that you wrote yourself, you should ensure that the `Node` implementation for each tag is thread-safe. For more information, see *template tag thread safety considerations*.

Test runner exit status code

The exit status code of the test runners (`tests/runtests.py` and `python manage.py test`) no longer represents the number of failed tests, since a failure of 256 or more tests resulted in a wrong exit status code. The exit status code for the test runner is now 0 for success (no failing tests) and 1 for any number of test failures. If needed, the number of test failures can be found at the end of the test runner's output.

Features deprecated in 1.2

CSRF response rewriting middleware

`CsrfResponseMiddleware`, the middleware that automatically inserted CSRF tokens into POST forms in outgoing pages, has been deprecated in favor of a template tag method (see above), and will be removed completely in Django 1.4. `CsrfMiddleware`, which includes the functionality of `CsrfResponseMiddleware` and `CsrfViewMiddleware` has likewise been deprecated.

Also, the CSRF module has moved from contrib to core, and the old imports are deprecated, as described in the upgrading notes.

Documentation removed

The upgrade notes have been removed in current Django docs. Please refer to the docs for Django 1.3 or older to find these instructions.

`SMTPConnection`

The `SMTPConnection` class has been deprecated in favor of a generic Email backend API. Old code that explicitly instantiated an instance of an SMTPConnection:

```
from django.core.mail import SMTPConnection
connection = SMTPConnection()
messages = get_notification_email()
connection.send_messages(messages)
```

should now call `get_connection()` to instantiate a generic email connection:

```
from django.core.mail import get_connection
connection = get_connection()
messages = get_notification_email()
connection.send_messages(messages)
```

Depending on the value of the `EMAIL_BACKEND` setting, this may not return an SMTP connection. If you explicitly require an SMTP connection with which to send email, you can explicitly request an SMTP connection:

```
from django.core.mail import get_connection
connection = get_connection('django.core.mail.backends.smtp.EmailBackend')
messages = get_notification_email()
connection.send_messages(messages)
```

If your call to construct an instance of `SMTPConnection` required additional arguments, those arguments can be passed to the `get_connection()` call:

```
connection = get_connection('django.core.mail.backends.smtp.EmailBackend', hostname='localhost', port
```

Specifying databases

Prior to Django 1.1, Django used a number of settings to control access to a single database. Django 1.2 introduces support for multiple databases, and as a result, the way you define database settings has changed.

Any existing Django settings file will continue to work as expected until Django 1.4. Old-style database settings will be automatically translated to the new-style format.

In the old-style (pre 1.2) format, there were a number of `DATABASE_` settings at the top level of your settings file. For example:

```
DATABASE_NAME = 'test_db'
DATABASE_ENGINE = 'postgresql_psycopg2'
DATABASE_USER = 'myusername'
DATABASE_PASSWORD = 's3krit'
```

These settings are now contained inside a dictionary named `DATABASES`. Each item in the dictionary corresponds to a single database connection, with the name `'default'` describing the default database connection. The setting names have also been shortened to reflect the fact that they are stored in a dictionary. The sample settings given previously would now be stored using:

```
DATABASES = {
    'default': {
        'NAME': 'test_db',
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'USER': 'myusername',
        'PASSWORD': 's3krit',
    }
}
```

This affects the following settings:

Old setting	New Setting
DATABASE_ENGINE	ENGINE
DATABASE_HOST	HOST
DATABASE_NAME	NAME
DATABASE_OPTIONS	OPTIONS
DATABASE_PASSWORD	PASSWORD
DATABASE_PORT	PORT
DATABASE_USER	USER
TEST_DATABASE_CHARSET	TEST_CHARSET
TEST_DATABASE_COLLATION	TEST_COLLATION
TEST_DATABASE_NAME	TEST_NAME

These changes are also required if you have manually created a database connection using `DatabaseWrapper()` from your database backend of choice.

In addition to the change in structure, Django 1.2 removes the special handling for the built-in database backends. All database backends must now be specified by a fully qualified module name (i.e., `django.db.backends.postgresql_psycopg2`, rather than just `postgresql_psycopg2`).

User Messages API

The API for storing messages in the user `Message` model (via `user.message_set.create`) is now deprecated and will be removed in Django 1.4 according to the standard *release process*.

To upgrade your code, you need to replace any instances of:

```
user.message_set.create('a message')
```

with the following:

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'a message')
```

Additionally, if you make use of the method, you need to replace the following:

```
for message in user.get_and_delete_messages():
    ...
```

with:

```
from django.contrib import messages
for message in messages.get_messages(request):
    ...
```

For more information, see the full *messages documentation*. You should begin to update your code to use the new API immediately.

Date format helper functions

`django.utils.translation.get_date_formats()` and `django.utils.translation.get_partial_date_fo` have been deprecated in favor of the appropriate calls to `django.utils.formats.get_format()` which is locale aware when `USE_L10N` is set to `True`, and falls back to default settings if set to `False`.

To get the different date formats, instead of writing:

```
from django.utils.translation import get_date_formats
date_format, datetime_format, time_format = get_date_formats()
```

use:

```
from django.utils import formats

date_format = formats.get_format('DATE_FORMAT')
datetime_format = formats.get_format('DATETIME_FORMAT')
time_format = formats.get_format('TIME_FORMAT')
```

or, when directly formatting a date value:

```
from django.utils import formats
value_formatted = formats.date_format(value, 'DATETIME_FORMAT')
```

The same applies to the globals found in `django.forms.fields`:

- `DEFAULT_DATE_INPUT_FORMATS`
- `DEFAULT_TIME_INPUT_FORMATS`
- `DEFAULT_DATETIME_INPUT_FORMATS`

Use `django.utils.formats.get_format()` to get the appropriate formats.

What's new in Django 1.2 alpha 1

The following new features are present as of this alpha release; this release also marks the end of major feature development for the 1.2 release cycle. Some minor features will continue development until the 1.2 beta release, however.

CSRF support

Django now has much improved protection against *Cross-Site Request Forgery (CSRF) attacks*. This type of attack occurs when a malicious Web site contains a link, a form button or some javascript that is intended to perform some action on your Web site, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, 'login CSRF', where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered.

Email Backends

You can now *configure the way that Django sends email*. Instead of using SMTP to send all email, you can now choose a configurable email backend to send messages. If your hosting provider uses a sandbox or some other non-SMTP technique for sending mail, you can now construct an email backend that will allow Django's standard *mail sending methods* to use those facilities.

This also makes it easier to debug mail sending - Django ships with backend implementations that allow you to send email to a *file*, to the *console*, or to *memory* - you can even configure all email to be *thrown away*.

Messages Framework

Django now includes a robust and configurable *messages framework* with built-in support for cookie- and session-based messaging, for both anonymous and authenticated clients. The messages framework replaces the deprecated user message API and allows you to temporarily store messages in one request and retrieve them for display in a subsequent request (usually the next one).

Support for multiple databases

Django 1.2 adds the ability to use *more than one database* in your Django project. Queries can be issued at a specific database with the *using()* method on querysets; individual objects can be saved to a specific database by providing a `using` argument when you save the instance.

'Smart' if tag

The `if` tag has been upgraded to be much more powerful. First, support for comparison operators has been added. No longer will you have to type:

```
{% ifnotequal a b %}
 ...
{% endifnotequal %}
```

...as you can now do:

```
{% if a != b %}
 ...
{% endif %}
```

The operators supported are ==, !=, <, >, <=, >= and `in`, all of which work like the Python operators, in addition to `and`, `or` and `not` which were already supported.

Also, filters may now be used in the `if` expression. For example:

```
<div
  {% if user.email|lower == message.recipient|lower %}
    class="highlight"
  {% endif %}
>{{ message }}</div>
```

Template caching

In previous versions of Django, every time you rendered a template it would be reloaded from disk. In Django 1.2, you can use a *cached template loader* to load templates once, then use the cached result for every subsequent render. This can lead to a significant performance improvement if your templates are broken into lots of smaller subtemplates (using the `{% extends %}` or `{% include %}` tags).

As a side effect, it is now much easier to support non-Django template languages. For more details, see the *notes on supporting non-Django template languages*.

Natural keys in fixtures

Fixtures can refer to remote objects using *Natural keys*. This lookup scheme is an alternative to the normal primary-key based object references in a fixture, improving readability, and resolving problems referring to objects whose primary key value may not be predictable or known.

`BigIntegerField`

Models can now use a 64 bit `BigIntegerField` type.

Fast Failure for Tests

The `test` subcommand of `django-admin.py`, and the `runtests.py` script used to run Django's own test suite, support a new `--failfast` option. When specified, this option causes the test runner to exit after encountering a failure instead of continuing with the test run. In addition, the handling of `Ctrl-C` during a test run has been improved to trigger a graceful exit from the test run that reports details of the tests run before the interruption.

Improved localization

Django's *internationalization framework* has been expanded by locale aware formatting and form processing. That means, if enabled, dates and numbers on templates will be displayed using the format specified for the current locale. Django will also use localized formats when parsing data in forms. See *Format localization* for more details.

Added `readonly_fields` to `ModelAdmin`

`django.contrib.admin.ModelAdmin.readonly_fields` has been added to enable non-editable fields in add/change pages for models and inlines. Field and calculated values can be displayed along side editable fields.

Customizable syntax highlighting

You can now use the `DJANGO_COLORS` environment variable to modify or disable the colors used by `django-admin.py` to provide *syntax highlighting*.

The Django 1.2 roadmap

Before the final Django 1.2 release, several other preview/development releases will be made available. The current schedule consists of at least the following:

- Week of **January 26, 2010**: First Django 1.2 beta release. Final feature freeze for Django 1.2.
- Week of **March 2, 2010**: First Django 1.2 release candidate. String freeze for translations.
- Week of **March 9, 2010**: Django 1.2 final release.

If necessary, additional alpha, beta or release-candidate packages will be issued prior to the final 1.2 release. Django 1.2 will be released approximately one week after the final release candidate.

What you can do to help

In order to provide a high-quality 1.2 release, we need your help. Although this alpha release is, again, *not* intended for production use, you can help the Django team by trying out the alpha codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.2 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Development sprints for Django 1.2 will also be taking place at PyCon US 2010, on the dedicated sprint days (February 22 through 25), and anyone who wants to help out is welcome to join in, either in person at PyCon or virtually in the IRC channel or on the mailing list.

9.2.8 Django 1.1 RC 1 release notes

July 21, 2009

Welcome to the first Django 1.1 release candidate!

This is the third – and likely last – in a series of preview/development releases leading up to the eventual release of Django 1.1, currently scheduled to take place approximately one week after this release candidate. This release is targeted primarily at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve any critical bugs prior to the final 1.1 release.

As such, this release is not yet intended for production use, and any such use is discouraged.

What's new in Django 1.1 RC 1

The Django codebase has – with one exception – been in feature freeze since the first 1.1 beta release, and so this release candidate contains only one new feature (see below); work leading up to this release candidate has instead been focused on bugfixing, particularly on the new features introduced prior to the 1.1 beta.

For an overview of those features, consult *the Django 1.1 beta release notes*.

URL namespaces

The 1.1 beta release introduced the ability to use reverse URL resolution with Django's admin application, which exposed a set of *named URLs*. Unfortunately, achieving consistent and correct reverse resolution for admin URLs proved extremely difficult, and so one additional feature was added to Django to resolve this issue: URL namespaces.

In short, this feature allows the same group of URLs, from the same application, to be included in a Django URLConf multiple times, with varying (and potentially nested) named prefixes which will be used when performing reverse resolution. For full details, see *the documentation on defining URL namespaces*.

Due to the changes needed to support this feature, the URL pattern names used when reversing admin URLs have changed since the 1.1 beta release; if you were developing applications which took advantage of this new feature, you will need to update your code to reflect the new names (for most purposes, changing `admin_` to `admin:` in names to be reversed will suffice). For a full list of URL pattern names used by the admin and information on how namespaces are applied to them, consult the documentation on *reversing admin URLs*.

The Django 1.1 roadmap

As of this release candidate, Django 1.1 is in both feature freeze and "string freeze" – all strings marked for translation in the Django codebase will retain their current form in the final Django 1.1 release. Only critical release-blocking bugs will receive attention between now and the final 1.1 release.

If no such bugs are discovered, Django 1.1 will be released approximately one week after this release candidate, on or about July 28, 2009.

What you can do to help

In order to provide a high-quality 1.1 release, we need your help. Although this release candidate is, again, *not* intended for production use, you can help the Django team by trying out this release candidate in a safe testing environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open a new ticket only if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.1 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

9.2.9 Django 1.1 beta 1 release notes

March 23, 2009

Welcome to Django 1.1 beta 1!

This is the second in a series of preview/development releases leading up to the eventual release of Django 1.1, currently scheduled to take place in April 2009. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.1 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

What's new in Django 1.1 beta 1

See Also:

The *1.1 alpha release notes*, which has a list of everything new between Django 1.0 and Django 1.1 alpha.

Model improvements

A number of features have been added to Django's model layer:

"Unmanaged" models You can now control whether or not Django creates database tables for a model using the `managed` model option. This defaults to `True`, meaning that Django will create the appropriate database tables in `syncdb` and remove them as part of `reset` command. That is, Django *manages* the database table's lifecycle.

If you set this to `False`, however, no database table creating or deletion will be automatically performed for this model. This is useful if the model represents an existing table or a database view that has been created by some other means.

For more details, see the documentation for the `managed` option.

Proxy models You can now create *proxy models*: subclasses of existing models that only add Python behavior and aren't represented by a new table. That is, the new model is a *proxy* for some underlying model, which stores all the real data.

All the details can be found in the *proxy models documentation*. This feature is similar on the surface to unmanaged models, so the documentation has an explanation of *how proxy models differ from unmanaged models*.

Deferred fields In some complex situations, your models might contain fields which could contain a lot of data (for example, large text fields), or require expensive processing to convert them to Python objects. If you know you don't need those particular fields, you can now tell Django not to retrieve them from the database.

You'll do this with the new queryset methods `defer()` and `only()`.

New admin features

Since 1.1 alpha, a couple of new features have been added to Django's admin application:

Editable fields on the change list You can now make fields editable on the admin list views via the new *list_editable* admin option. These fields will show up as form widgets on the list pages, and can be edited and saved in bulk.

Admin "actions" You can now define *admin actions* that can perform some action to a group of models in bulk. Users will be able to select objects on the change list page and then apply these bulk actions to all selected objects.

Django ships with one pre-defined admin action to delete a group of objects in one fell swoop.

Testing improvements

A couple of small but very useful improvements have been made to the *testing framework*:

- The test `Client` now can automatically follow redirects with the `follow` argument to `Client.get()` and `Client.post()`. This makes testing views that issue redirects simpler.
- It's now easier to get at the template context in the response returned the test client: you'll simply access the context as `request.context[key]`. The old way, which treats `request.context` as a list of contexts, one for each rendered template, is still available if you need it.

Conditional view processing

Django now has much better support for *conditional view processing* using the standard `ETag` and `Last-Modified` HTTP headers. This means you can now easily short-circuit view processing by testing less-expensive conditions. For many views this can lead to a serious improvement in speed and reduction in bandwidth.

Other improvements

Finally, a grab-bag of other neat features made their way into this beta release, including:

- The `dumpdata` management command now accepts individual model names as arguments, allowing you to export the data just from particular models.

- There's a new `safeseq` template filter which works just like `safe` for lists, marking each item in the list as safe.
- *Cache backends* now support `incr()` and `decr()` commands to increment and decrement the value of a cache key. On cache backends that support atomic increment/decrement – most notably, the memcached backend – these operations will be atomic, and quite fast.
- Django now can *easily delegate authentication to the Web server* via a new authentication backend that supports the standard `REMOTE_USER` environment variable used for this purpose.
- There's a new `django.shortcuts.redirect()` function that makes it easier to issue redirects given an object, a view name, or a URL.
- The `postgresql_psycopg2` backend now supports *native PostgreSQL autocommit*. This is an advanced, PostgreSQL-specific feature, that can make certain read-heavy applications a good deal faster.

The Django 1.1 roadmap

Before Django 1.1 goes final, at least one other preview/development release will be made available. The current schedule consists of at least the following:

- Week of *April 2, 2009:* Django 1.1 release candidate. At this point all strings marked for translation must freeze to allow translations to be submitted in advance of the final release.
- Week of *April 13, 2009:* Django 1.1 final.

If deemed necessary, additional beta or release candidate packages will be issued prior to the final 1.1 release.

What you can do to help

In order to provide a high-quality 1.1 release, we need your help. Although this beta release is, again, *not* intended for production use, you can help the Django team by trying out the beta codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.1 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Development sprints for Django 1.1 will also be taking place at PyCon US 2009, on the dedicated sprint days (March 30 through April 2), and anyone who wants to help out is welcome to join in, either in person at PyCon or virtually in the IRC channel or on the mailing list.

9.2.10 Django 1.1 alpha 1 release notes

February 23, 2009

Welcome to Django 1.1 alpha 1!

This is the first in a series of preview/development releases leading up to the eventual release of Django 1.1, currently scheduled to take place in April 2009. This release is primarily targeted at developers who are interested in trying out new features and testing the Django codebase to help identify and resolve bugs prior to the final 1.1 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

What's new in Django 1.1 alpha 1

ORM improvements

Two major enhancements have been added to Django's object-relational mapper (ORM):

Aggregate support It's now possible to run SQL aggregate queries (i.e. `COUNT()`, `MAX()`, `MIN()`, etc.) from within Django's ORM. You can choose to either return the results of the aggregate directly, or else annotate the objects in a `QuerySet` with the results of the aggregate query.

This feature is available as new `QuerySet.aggregate()`‘() and `QuerySet.annotate()`‘() methods, and is covered in detail in *the ORM aggregation documentation*

Query expressions Queries can now refer to a another field on the query and can traverse relationships to refer to fields on related models. This is implemented in the new `F` object; for full details, including examples, consult the *documentation for F expressions*.

Performance improvements

Tests written using Django's *testing framework* now run dramatically faster (as much as 10 times faster in many cases).

This was accomplished through the introduction of transaction-based tests: when using `django.test.TestCase`, your tests will now be run in a transaction which is rolled back when finished, instead of by flushing and re-populating the database. This results in an immense speedup for most types of unit tests. See the documentation for `TestCase` and `TransactionTestCase` for a full description, and some important notes on database support.

Other improvements

Other new features and changes introduced since Django 1.0 include:

- The *CSRF protection middleware* has been split into two classes – `CsrfViewMiddleware` checks incoming requests, and `CsrfResponseMiddleware` processes outgoing responses. The combined `CsrfMiddleware` class (which does both) remains for backwards-compatibility, but using the split classes is now recommended in order to allow fine-grained control of when and where the CSRF processing takes place.
- `reverse()` and code which uses it (e.g., the `{% url %}` template tag) now works with URLs in Django's administrative site, provided that the admin URLs are set up via `include(admin.site.urls)` (sending admin requests to the `admin.site.root` view still works, but URLs in the admin will not be "reversible" when configured this way).
- The `include()` function in Django URLconf modules can now accept sequences of URL patterns (generated by `patterns()`) in addition to module names.

- Instances of Django forms (see *the forms overview*) now have two additional methods, `hidden_fields()` and `visible_fields()`, which return the list of hidden – i.e., `<input type="hidden">` – and visible fields on the form, respectively.
- The `redirect_to` generic view now accepts an additional keyword argument `permanent`. If `permanent` is `True`, the view will emit an HTTP permanent redirect (status code 301). If `False`, the view will emit an HTTP temporary redirect (status code 302).
- A new database lookup type – `week_day` – has been added for `DateField` and `DateTimeField`. This type of lookup accepts a number between 1 (Sunday) and 7 (Saturday), and returns objects where the field value matches that day of the week. See *the full list of lookup types* for details.
- The `{% for %}` tag in Django's template language now accepts an optional `{% empty %}` clause, to be displayed when `{% for %}` is asked to loop over an empty sequence. See *the list of built-in template tags* for examples of this.

The Django 1.1 roadmap

Before Django 1.1 goes final, several other preview/development releases will be made available. The current schedule consists of at least the following:

- Week of *March 20, 2009:* Django 1.1 beta 1, at which point Django 1.1 will be in "feature freeze": no new features will be implemented for 1.1 past that point, and all new feature work will be deferred to Django 1.2.
- Week of *April 2, 2009:* Django 1.1 release candidate. At this point all strings marked for translation must freeze to allow translations to be submitted in advance of the final release.
- Week of *April 13, 2009:* Django 1.1 final.

If deemed necessary, additional alpha, beta or release candidate packages will be issued prior to the final 1.1 release.

What you can do to help

In order to provide a high-quality 1.1 release, we need your help. Although this alpha release is, again, *not* intended for production use, you can help the Django team by trying out the alpha codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.1 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

... and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *How to contribute to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Development sprints for Django 1.1 will also be taking place at PyCon US 2009, on the dedicated sprint days (March 30 through April 2), and anyone who wants to help out is welcome to join in, either in person at PyCon or virtually in the IRC channel or on the mailing list.

9.2.11 Django 1.0 beta 2 release notes

Welcome to Django 1.0 beta 2!

This is the fourth in a series of preview/development releases leading up to the eventual release of Django 1.0, currently scheduled to take place in early September 2008. This releases is primarily targeted at developers who are interested in testing the Django codebase and helping to identify and resolve bugs prior to the final 1.0 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

What's new in Django 1.0 beta 2

Django's development trunk has been the site of nearly constant activity over the past year, with several major new features landing since the 0.96 release. For features which were new as of Django 1.0 alpha 1, see *the 1.0 alpha 1 release notes*. For features which were new as of Django 1.0 alpha 2, see *the 1.0 alpha 2 release notes*. For features which were new as of Django 1.0 beta 1, see *the 1.0 beta 1 release notes*.

This beta release includes two major features:

Refactored `django.contrib.comments` As part of a Google Summer of Code project, Thejaswi Puthraya carried out a major rewrite and refactoring of Django's bundled comment system, greatly increasing its flexibility and customizability. *Full documentation* is available, as well as *an upgrade guide* if you were using the previous incarnation of the comments application..

Refactored documentation Django's bundled and online documentation has also been significantly refactored; the new documentation system uses Sphinx to build the docs and handle such niceties as topical indexes, reference documentation and cross-references within the docs. You can check out the new documentation online or, if you have Sphinx installed, build the HTML yourself from the documentation files bundled with Django.

Along with these new features, the Django team has also been hard at work polishing Django's codebase for the final 1.0 release; this beta release contains a large number of smaller improvements and bugfixes from the ongoing push to 1.0.

Also, as part of its ongoing deprecation process, Django's old form-handling system has been removed; this means `django.oldforms` no longer exists, and its various API hooks (such as automatic manipulators) are no longer present in Django. This system has been completely replaced by *the new form-handling system* in `django.forms`.

The Django 1.0 roadmap

One of the primary goals of this beta release is to focus attention on the remaining features to be implemented for Django 1.0, and on the bugs that need to be resolved before the final release. As of this beta release, Django is in its final "feature freeze" for 1.0; feature requests will be deferred to later releases, and the development effort will be focused solely on bugfixing and stability. Django is also now in a "string freeze"; translatable strings (labels, error messages, etc.) in Django's codebase will not be changed prior to the release, in order to allow our translators to produce the final 1.0 version of Django's translation files.

Following this release, we'll be conducting a final development sprint on August 30, 2008, based in London and coordinated online; the goal of this sprint will be to squash as many bugs as possible in anticipation of the final 1.0 release, which is currently targeted for **September 2, 2008**. The official Django 1.0 release party will take place during the first-ever DjangoCon, to be held in Mountain View, California, USA, September 6-7.

What you can do to help

In order to provide a high-quality 1.0 release, we need your help. Although this beta release is, again, *not* intended for production use, you can help the Django team by trying out the beta codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.0 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

...and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *contributing to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

9.2.12 Django 1.0 beta 1 release notes

Welcome to Django 1.0 beta 1!

This is the third in a series of preview/development releases leading up to the eventual release of Django 1.0, currently scheduled to take place in early September 2008. This releases is primarily targeted at developers who are interested in testing the Django codebase and helping to identify and resolve bugs prior to the final 1.0 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

What's new in Django 1.0 beta 1

Django's development trunk has been the site of nearly constant activity over the past year, with several major new features landing since the 0.96 release. For features which were new as of Django 1.0 alpha 1, see *the 1.0 alpha 1 release notes*. For features which were new as of Django 1.0 alpha 2, see *the 1.0 alpha 2 release notes*.

This beta release does not contain any major new features, but does include several smaller updates and improvements to Django:

Generic relations in forms and admin Classes are now included in `django.contrib.contenttypes` which can be used to support generic relations in both the admin interface and in end-user forms. See *the documentation for generic relations* for details.

Improved flexibility in the admin Following up on the refactoring of Django's administrative interface (`django.contrib.admin`), introduced in Django 1.0 alpha 1, two new hooks have been added to allow customized pre- and post-save handling of model instances in the admin. Full details are in *the admin documentation*.

`INSERT/UPDATE` distinction Although Django's default behavior of having a model's `save()` method automatically determine whether to perform an `INSERT` or an `UPDATE` at the SQL level is suitable for the majority of cases, there are occasional situations where forcing one or the other is useful. As a result, models can now support an additional parameter to `save()` which can force a specific operation. Consult the database API documentation for details and important notes about appropriate use of this parameter.

Split `CacheMiddleware` Django's `CacheMiddleware` has been split into three classes: `CacheMiddleware` itself still exists and retains all of its previous functionality, but it is now built from two separate middleware classes which handle the two parts of caching (inserting into and reading from the cache) separately, offering additional flexibility for situations where combining these functions into a single middleware posed problems. Full details, including updated notes on appropriate use, are in *the caching documentation*.

Removal of deprecated features A number of features and methods which had previously been marked as deprecated, and which were scheduled for removal prior to the 1.0 release, are no longer present in Django. These include imports of the form library from `django.newforms` (now located simply at `django.forms`), the `form_for_model` and `form_for_instance` helper functions (which have been replaced by `ModelForm`) and a number of deprecated features which were replaced by the dispatcher, file-uploading and file-storage refactorings introduced in the Django 1.0 alpha releases. A full list of these and all other backwards-incompatible changes is available on the Django wiki.

A number of other improvements and bugfixes have also been included: some tricky cases involving case-sensitivity in differing MySQL collations have been resolved, Windows packaging and installation has been improved and the method by which Django generates unique session identifiers has been made much more robust.

The Django 1.0 roadmap

One of the primary goals of this beta release is to focus attention on the remaining features to be implemented for Django 1.0, and on the bugs that need to be resolved before the final release. Following this release, we'll be conducting a series of development sprints building up to the release-candidate stage, followed soon after by Django 1.0. The timeline is projected to be:

- August 15, 2008: Sprint (based in Austin, Texas, USA, and online).
- August 17, 2008: Sprint (based in Tel Aviv, Israel, and online).
- **August 21, 2008: Django 1.0 release candidate 1.** At this point, all strings marked for translation within Django's codebase will be frozen, to provide contributors time to check and finalize all of Django's bundled translation files prior to the final 1.0 release.
- August 22, 2008: Sprint (based in Portland, Oregon, USA, and online).
- **August 26, 2008: Django 1.0 release candidate 2.**
- August 30, 2008: Sprint (based in London, England, UK, and online).
- **September 2, 2008: Django 1.0 final release.** The official Django 1.0 release party will take place during the first-ever DjangoCon, to be held in Mountain View, California, USA, September 6-7.

Of course, like any estimated timeline, this is subject to change as requirements dictate. The latest information will always be available on the Django project wiki:

- https://code.djangoproject.com/wiki/VersionOneRoadmap

What you can do to help

In order to provide a high-quality 1.0 release, we need your help. Although this beta release is, again, *not* intended for production use, you can help the Django team by trying out the beta codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.0 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

...and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *contributing to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

9.2.13 Django 1.0 alpha 2 release notes

Welcome to Django 1.0 alpha 2!

This is the second in a series of preview/development releases leading up to the eventual release of Django 1.0, currently scheduled to take place in early September 2008. This releases is primarily targeted at developers who are interested in testing the Django codebase and helping to identify and resolve bugs prior to the final 1.0 release.

As such, this release is *not* intended for production use, and any such use is strongly discouraged.

What's new in Django 1.0 alpha 2

Django's development trunk has been the site of nearly constant activity over the past year, with several major new features landing since the 0.96 release. For features which were new as of Django 1.0 alpha 1, see *the 1.0 alpha 1 release notes*. Since the 1.0 alpha 1 release several new features have landed, including:

`django.contrib.gis` (GeoDjango) A project over a year in the making, this adds world-class GIS (Geographic Information Systems) support to Django, in the form of a `contrib` application. Its documentation is currently being maintained externally, and will be merged into the main Django documentation prior to the final 1.0 release. Huge thanks go to Justin Bronn, Jeremy Dunck, Brett Hoerner and Travis Pinney for their efforts in creating and completing this feature.

Pluggable file storage Django's built-in `FileField` and `ImageField` now can take advantage of pluggable file-storage backends, allowing extensive customization of where and how uploaded files get stored by Django. For details, see *the files documentation*; big thanks go to Marty Alchin for putting in the hard work to get this completed.

Jython compatibility Thanks to a lot of work from Leo Soto during a Google Summer of Code project, Django's codebase has been refactored to remove incompatibilities with Jython, an implementation of Python written in Java, which runs Python code on the Java Virtual Machine. Django is now compatible with the forthcoming Jython 2.5 release.

There are many other new features and improvements in this release, including two major performance boosts: strings marked for translation using *Django's internationalization system* now consume far less memory, and Django's internal dispatcher – which is invoked frequently during request/response processing and when working with Django's object-relational mapper – is now significantly faster.

The Django 1.0 roadmap

One of the primary goals of this alpha release is to focus attention on the remaining features to be implemented for Django 1.0, and on the bugs that need to be resolved before the final release. Following this release, we'll be conducting a series of development sprints building up to the beta and release-candidate stages, followed soon after by Django 1.0. The timeline is projected to be:

- **August 14, 2008: Django 1.0 beta release.** Past this point Django will be in a "feature freeze" for the 1.0 release; after Django 1.0 beta, the development focus will be solely on bug fixes and stabilization.
- August 15, 2008: Sprint (based in Austin, Texas, USA, and online).
- August 17, 2008: Sprint (based in Tel Aviv, Israel, and online).

- **August 21, 2008: Django 1.0 release candidate 1.** At this point, all strings marked for translation within Django's codebase will be frozen, to provide contributors time to check and finalize all of Django's bundled translation files prior to the final 1.0 release.
- August 22, 2008: Sprint (based in Portland, Oregon, USA, and online).
- **August 26, 2008: Django 1.0 release candidate 2.**
- August 30, 2008: Sprint (based in London, England, UK, and online).
- **September 2, 2008: Django 1.0 final release.** The official Django 1.0 release party will take place during the first-ever DjangoCon, to be held in Mountain View, California, USA, September 6-7.

Of course, like any estimated timeline, this is subject to change as requirements dictate. The latest information will always be available on the Django project wiki:

- https://code.djangoproject.com/wiki/VersionOneRoadmap

What you can do to help

In order to provide a high-quality 1.0 release, we need your help. Although this alpha release is, again, *not* intended for production use, you can help the Django team by trying out the alpha codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.0 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

...and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *contributing to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

9.2.14 Django 1.0 alpha release notes

Welcome to Django 1.0 alpha!

This is the first in a series of preview/development releases leading up to the eventual release of Django 1.0, currently scheduled to take place in early September 2008. This release is primarily targeted at developers who are interested in testing the Django codebase and helping to identify and resolve bugs prior to the final 1.0 release.

As such, this release is *not* intended for production use, and any such use is strongly discouraged.

What's new in Django 1.0 alpha

Django's development trunk has been the site of nearly constant activity over the past year, with several major new features landing since the 0.96 release. Some of the highlights include:

Refactored admin application (newforms-admin) The Django administrative interface (`django.contrib.admin`) has been completely refactored; admin definitions are now completely decoupled from model definitions (no more `class Admin` declaration in models!), rewritten to use Django's new form-handling library (introduced in the 0.96 release as `django.newforms`, and now available as simply `django.forms`) and redesigned with extensibility and customization in mind. Full documentation for the admin application is available online in the official Django documentation:

- *admin reference*

Improved Unicode handling Django's internals have been refactored to use Unicode throughout; this drastically simplifies the task of dealing with non-Western-European content and data in Django. Additionally, utility functions have been provided to ease interoperability with third-party libraries and systems which may or may not handle Unicode gracefully. Details are available in Django's Unicode-handling documentation:

- *unicode reference*

An improved Django ORM Django's object-relational mapper – the component which provides the mapping between Django model classes and your database, and which mediates your database queries – has been dramatically improved by a massive refactoring. For most users of Django this is backwards-compatible; the public-facing API for database querying underwent a few minor changes, but most of the updates took place in the ORM's internals. A guide to the changes, including backwards-incompatible modifications and mentions of new features opened up by this refactoring, is available on the Django wiki:

- https://code.djangoproject.com/wiki/QuerysetRefactorBranch

Automatic escaping of template variables To provide improved security against cross-site scripting (XSS) vulnerabilities, Django's template system now automatically escapes the output of variables. This behavior is configurable, and allows both variables and larger template constructs to be marked as safe (requiring no escaping) or unsafe (requiring escaping). A full guide to this feature is in the documentation for the `autoescape` tag.

There are many more new features, many bugfixes and many enhancements to existing features from previous releases. The `newforms` library, for example, has undergone massive improvements including several useful add-ons in `django.contrib` which complement and build on Django's form-handling capabilities, and Django's file-uploading handlers have been refactored to allow finer-grained control over the uploading process as well as streaming uploads of large files.

Along with these improvements and additions, we've made a number of of backwards-incompatible changes to the framework, as features have been fleshed out and APIs have been finalized for the 1.0 release. A complete guide to these changes will be available as part of the final Django 1.0 release, and a comprehensive list of backwards-incompatible changes is also available on the Django wiki for those who want to begin developing and testing their upgrade process:

- https://code.djangoproject.com/wiki/BackwardsIncompatibleChanges

The Django 1.0 roadmap

One of the primary goals of this alpha release is to focus attention on the remaining features to be implemented for Django 1.0, and on the bugs that need to be resolved before the final release. Following this release, we'll be conducting a series of sprints building up to a series of beta releases and a release-candidate stage, followed soon after by Django 1.0. The timeline is projected to be:

- August 1, 2008: Sprint (based in Washington, DC, and online).
- August 5, 2008: Django 1.0 beta 1 release. This will also constitute the feature freeze for 1.0. Any feature to be included in 1.0 must be completed and in trunk by this time.
- August 8, 2008: Sprint (based in Lawrence, KS, and online).
- August 12, 2008: Django 1.0 beta 2 release.

- August 15, 2008: Sprint (based in Austin, TX, and online).
- August 19, 2008: Django 1.0 release candidate 1.
- August 22, 2008: Sprint (based in Portland, OR, and online).
- August 26, 2008: Django 1.0 release candidate 2.
- September 2, 2008: Django 1.0 final release. The official Django 1.0 release party will take place during the first-ever DjangoCon, to be held in Mountain View, CA, September 6-7.

Of course, like any estimated timeline, this is subject to change as requirements dictate. The latest information will always be available on the Django project wiki:

- https://code.djangoproject.com/wiki/VersionOneRoadmap

What you can do to help

In order to provide a high-quality 1.0 release, we need your help. Although this alpha release is, again, *not* intended for production use, you can help the Django team by trying out the alpha codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

- https://code.djangoproject.com/timeline

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.0 release, takes place daily on the django-developers mailing list:

- http://groups.google.com/group/django-developers

...and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

- *contributing to Django*

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

CHAPTER

TEN

DJANGO INTERNALS

Documentation for people hacking on Django itself. This is the place to go if you'd like to help improve Django, learn or learn about how Django works "under the hood".

Warning: Elsewhere in the Django documentation, coverage of a feature is a sort of a contract: once an API is in the official documentation, we consider it "stable" and don't change it without a good reason. APIs covered here, however, are considered "internal-only": we reserve the right to change these internals if we must.

10.1 Contributing to Django

Django is a community that lives on its volunteers. As it keeps growing, we always need more people to help others. As soon as you learn Django, you can contribute in many ways:

- Join the django-users mailing list and answer questions. This mailing list has a huge audience, and we really want to maintain a friendly and helpful atmosphere. If you're new to the Django community, you should read the posting guidelines.
- Join the #django IRC channel on Freenode and answer questions. By explaining Django to other users, you're going to learn a lot about the framework yourself.
- Blog about Django. We syndicate all the Django blogs we know about on the community page; if you'd like to see your blog on that page you can register it here.
- Contribute to open-source Django projects, write some documentation, or release your own code as an open-source pluggable application. The ecosystem of pluggable applications is a big strength of Django, help us build it!

If you think working *with* Django is fun, wait until you start working *on* it. We're passionate about helping Django users make the jump to contributing members of the community, so there are several ways you can help Django's development:

- *Report bugs* in our ticket tracker.
- Join the django-developers mailing list and share your ideas for how to improve Django. We're always open to suggestions.
- *Submit patches* for new and/or fixed behavior. If you're looking for an easy way to start contributing to Django have a look at the easy pickings tickets.
- *Improve the documentation* or *write unit tests*.
- *Triage tickets and review patches* created by other users.

Really, **ANYONE** can do something to help make Django better and greater!

Browse the following sections to find out how:

10.1.1 Advice for new contributors

New contributor and not sure what to do? Want to help but just don't know how to get started? This is the section for you.

First steps

Start with these easy tasks to discover Django's development process.

- **Triage tickets**

  If an unreviewed ticket reports a bug, try and reproduce it. If you can reproduce it and it seems valid, make a note that you confirmed the bug and accept the ticket. Make sure the ticket is filed under the correct component area. Consider writing a patch that adds a test for the bug's behavior, even if you don't fix the bug itself. See more at *How can I help with triaging?*

- **Look for tickets that are accepted and review patches to build familiarity with the codebase and the process**

  Mark the appropriate flags if a patch needs docs or tests. Look through the changes a patch makes, and keep an eye out for syntax that is incompatible with older but still supported versions of Python. Run the tests and make sure they pass on your system. Where possible and relevant, try them out on a database other than SQLite. Leave comments and feedback!

- **Keep old patches up to date**

  Oftentimes the codebase will change between a patch being submitted and the time it gets reviewed. Make sure it still applies cleanly and functions as expected. Simply updating a patch is both useful and important! See more on *Submitting patches*.

- **Write some documentation**

  Django's documentation is great but it can always be improved. Did you find a typo? Do you think that something should be clarified? Go ahead and suggest a documentation patch! See also the guide on *Writing documentation*, in particular the tips for *Improving the documentation*.

Note: The reports page contains links to many useful Trac queries, including several that are useful for triaging tickets and reviewing patches as suggested above.

Guidelines

As a newcomer on a large project, it's easy to experience frustration. Here's some advice to make your work on Django more useful and rewarding.

- **Pick a subject area that you care about, that you are familiar with, or that you want to learn about**

  You don't already have to be an expert on the area you want to work on; you become an expert through your ongoing contributions to the code.

- **Analyze tickets' context and history**

  Trac isn't an absolute; the context is just as important as the words. When reading Trac, you need to take into account who says things, and when they were said. Support for an idea two years ago doesn't necessarily mean

that the idea will still have support. You also need to pay attention to who *hasn't* spoken – for example, if a core team member hasn't been recently involved in a discussion, then a ticket may not have the support required to get into trunk.

- **Start small**

  It's easier to get feedback on a little issue than on a big one. See the easy pickings.

- **If you're going to engage in a big task, make sure that your idea has support first**

  This means getting someone else to confirm that a bug is real before you fix the issue, and ensuring that the core team supports a proposed feature before you go implementing it.

- **Be bold! Leave feedback!**

  Sometimes it can be scary to put your opinion out to the world and say "this ticket is correct" or "this patch needs work", but it's the only way the project moves forward. The contributions of the broad Django community ultimately have a much greater impact than that of the core developers. We can't do it without YOU!

- **Err on the side of caution when marking things Ready For Check-in**

  If you're really not certain if a ticket is ready, don't mark it as such. Leave a comment instead, letting others know your thoughts. If you're mostly certain, but not completely certain, you might also try asking on IRC to see if someone else can confirm your suspicions.

- **Wait for feedback, and respond to feedback that you receive**

  Focus on one or two tickets, see them through from start to finish, and repeat. The shotgun approach of taking on lots of tickets and letting some fall by the wayside ends up doing more harm than good.

- **Be rigorous**

  When we say "**PEP 8**, and must have docs and tests", we mean it. If a patch doesn't have docs and tests, there had better be a good reason. Arguments like "I couldn't find any existing tests of this feature" don't carry much weight–while it may be true, that means you have the extra-important job of writing the very first tests for that feature, not that you get a pass from writing tests altogether.

FAQ

1. **This ticket I care about has been ignored for days/weeks/months! What can I do to get it committed?**

   First off, it's not personal. Django is entirely developed by volunteers (even the core developers), and sometimes folks just don't have time. The best thing to do is to send a gentle reminder to the django-developers mailing list asking for review on the ticket, or to bring it up in the #django-dev IRC channel.

2. **I'm sure my ticket is absolutely 100% perfect, can I mark it as RFC myself?**

   Short answer: No. It's always better to get another set of eyes on a ticket. If you're having trouble getting that second set of eyes, see question 1, above.

3. **My ticket has been in DDN forever! What should I do?**

   Design Decision Needed requires consensus about the right solution. At the very least it needs consensus among the core developers, and ideally it has consensus from the community as well. The best way to accomplish this is to start a thread on the django-developers mailing list, and for very complex issues to start a wiki page summarizing the problem and the possible solutions.

10.1.2 Reporting bugs and requesting features

Before reporting a bug or requesting a new feature, please consider these general points:

- Check that someone hasn't already filed the bug or feature request by searching or running custom queries in the ticket tracker.
- Don't use the ticket system to ask support questions. Use the django-users list or the #django IRC channel for that.
- Don't reopen issues that have been marked "wontfix" by a core developer. This mark means that the decision has been made that we can't or won't fix this particular issue. If you're not sure why, please ask on django-developers.
- Don't use the ticket tracker for lengthy discussions, because they're likely to get lost. If a particular ticket is controversial, please move the discussion to django-developers.

Reporting bugs

Well-written bug reports are *incredibly* helpful. However, there's a certain amount of overhead involved in working with any bug tracking system so your help in keeping our ticket tracker as useful as possible is appreciated. In particular:

- **Do** read the *FAQ* to see if your issue might be a well-known question.
- **Do** ask on django-users or #django *first* if you're not sure if what you're seeing is a bug.
- **Do** write complete, reproducible, specific bug reports. You must include a clear, concise description of the problem, and a set of instructions for replicating it. Add as much debug information as you can: code snippets, test cases, exception backtraces, screenshots, etc. A nice small test case is the best way to report a bug, as it gives us an easy way to confirm the bug quickly.
- **Don't** post to django-developers just to announce that you have filed a bug report. All the tickets are mailed to another list, django-updates, which is tracked by developers and interested community members; we see them as they are filed.

To understand the lifecycle of your ticket once you have created it, refer to *Triaging tickets*.

Reporting security issues

Important: Please report security issues **only** to security@djangoproject.com. This is a private list only open to long-time, highly trusted Django developers, and its archives are not publicly readable.

In the event of a confirmed vulnerability in Django itself, we will take the following actions:

- Acknowledge to the reporter that we've received the report and that a fix is forthcoming. We'll give a rough timeline and ask the reporter to keep the issue confidential until we announce it.
- Focus on developing a fix as quickly as possible and produce patches against the current and two previous releases.
- Determine a go-public date for announcing the vulnerability and the fix. To try to mitigate a possible "arms race" between those applying the patch and those trying to exploit the hole, we will not announce security problems immediately.
- Pre-notify third-party distributors of Django ("vendors"). We will send these vendor notifications through private email which will include documentation of the vulnerability, links to the relevant patch(es), and a request to keep the vulnerability confidential until the official go-public date.
- Publicly announce the vulnerability and the fix on the pre-determined go-public date. This will probably mean a new release of Django, but in some cases it may simply be patches against current releases.

Reporting user interface bugs and features

If your bug or feature request touches on anything visual in nature, there are a few additional guidelines to follow:

- Include screenshots in your ticket which are the visual equivalent of a minimal testcase. Show off the issue, not the crazy customizations you've made to your browser.
- If the issue is difficult to show off using a still image, consider capturing a *brief* screencast. If your software permits it, capture only the relevant area of the screen.
- If you're offering a patch which changes the look or behavior of Django's UI, you **must** attach before *and* after screenshots/screencasts. Tickets lacking these are difficult for triagers and core developers to assess quickly.
- Screenshots don't absolve you of other good reporting practices. Make sure to include URLs, code snippets, and step-by-step instructions on how to reproduce the behavior visible in the screenshots.
- Make sure to set the UI/UX flag on the ticket so interested parties can find your ticket.

Requesting features

We're always trying to make Django better, and your feature requests are a key part of that. Here are some tips on how to make a request most effectively:

- Make sure the feature actually requires changes in Django's core. If your idea can be developed as an independent application or module — for instance, you want to support another database engine — we'll probably suggest that you to develop it independently. Then, if your project gathers sufficient community support, we may consider it for inclusion in Django.
- First request the feature on the django-developers list, not in the ticket tracker. It'll get read more closely if it's on the mailing list. This is even more important for large-scale feature requests. We like to discuss any big changes to Django's core on the mailing list before actually working on them.
- Describe clearly and concisely what the missing feature is and how you'd like to see it implemented. Include example code (non-functional is OK) if possible.
- Explain *why* you'd like the feature. In some cases this is obvious, but since Django is designed to help real developers get real work done, you'll need to explain it, if it isn't obvious why the feature would be useful.

If core developers agree on the feature, then it's appropriate to create a ticket. Include a link the discussion on django-developers in the ticket description.

As with most open-source projects, code talks. If you are willing to write the code for the feature yourself or, even better, if you've already written it, it's much more likely to be accepted. Just fork Django on GitHub, create a feature branch, and show us your work!

See also: *Documenting new features*.

How we make decisions

Whenever possible, we strive for a rough consensus. To that end, we'll often have informal votes on django-developers about a feature. In these votes we follow the voting style invented by Apache and used on Python itself, where votes are given as +1, +0, -0, or -1. Roughly translated, these votes mean:

- +1: "I love the idea and I'm strongly committed to it."
- +0: "Sounds OK to me."
- -0: "I'm not thrilled, but I won't stand in the way."
- -1: "I strongly disagree and would be very unhappy to see the idea turn into reality."

Although these votes on django-developers are informal, they'll be taken very seriously. After a suitable voting period, if an obvious consensus arises we'll follow the votes.

However, consensus is not always possible. If consensus cannot be reached, or if the discussion towards a consensus fizzles out without a concrete decision, we use a more formal process.

Any *core committer* may call for a formal vote using the same voting mechanism above. A proposition will be considered carried by the core team if:

- There are three "+1" votes from members of the core team.
- There is no "-1" vote from any member of the core team.
- The *BDFLs* haven't stepped in and executed their positive or negative veto.

When calling for a vote, the caller should specify a deadline by which votes must be received. One week is generally suggested as the minimum amount of time.

Since this process allows any core committer to veto a proposal, any "-1" votes (or BDFL vetos) should be accompanied by an explanation that explains what it would take to convert that "-1" into at least a "+0".

Whenever possible, these formal votes should be announced and held in public on the django-developers mailing list. However, overly sensitive or contentious issues – including, most notably, votes on new core committers – may be held in private.

10.1.3 Triaging tickets

Django uses Trac for managing the work on the code base. Trac is a community-tended garden of the bugs people have found and the features people would like to see added. As in any garden, sometimes there are weeds to be pulled and sometimes there are flowers and vegetables that need picking. We need your help to sort out one from the other, and in the end we all benefit together.

Like all gardens, we can aspire to perfection but in reality there's no such thing. Even in the most pristine garden there are still snails and insects. In a community garden there are also helpful people who – with the best of intentions – fertilize the weeds and poison the roses. It's the job of the community as a whole to self-manage, keep the problems to a minimum, and educate those coming into the community so that they can become valuable contributing members.

Similarly, while we aim for Trac to be a perfect representation of the state of Django's progress, we acknowledge that this simply will not happen. By distributing the load of Trac maintenance to the community, we accept that there will be mistakes. Trac is "mostly accurate", and we give allowances for the fact that sometimes it will be wrong. That's okay. We're perfectionists with deadlines.

We rely on the community to keep participating, keep tickets as accurate as possible, and raise issues for discussion on our mailing lists when there is confusion or disagreement.

Django is a community project, and every contribution helps. We can't do this without YOU!

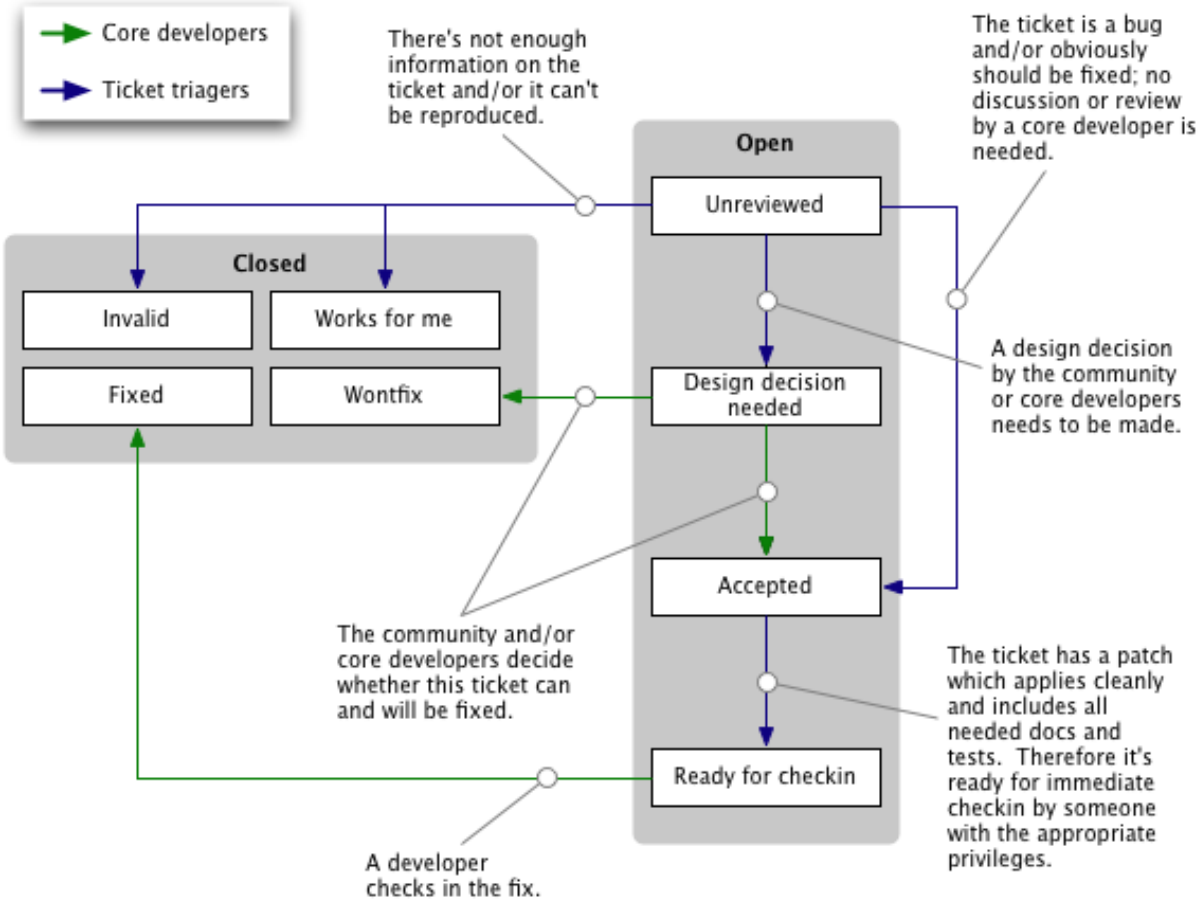
Triage workflow

Unfortunately, not all bug reports and feature requests in the ticket tracker provide all the *required details*. A number of tickets have patches, but those patches don't meet all the requirements of a *good patch*.

One way to help out is to *triage* tickets that have been created by other users. The core team and several community members work on this regularly, but more help is always appreciated.

Most of the workflow is based around the concept of a ticket's *triage stages*. Each stage describes where in its lifetime a given ticket is at any time. Along with a handful of flags, this attribute easily tells us what and who each ticket is waiting on.

Since a picture is worth a thousand words, let's start there:



We've got two roles in this diagram:

- *Committers* (also called core developers): people with commit access who are responsible for making the big decisions, writing large portions of the code and integrating the contributions of the community.
- Ticket triagers: anyone in the Django community who chooses to become involved in Django's development process. Our Trac installation is intentionally left open to the public, and anyone can triage tickets. Django is a community project, and we encourage *triage by the community*.

By way of example, here we see the lifecycle of an average ticket:

- Alice creates a ticket, and uploads an incomplete patch (no tests, incorrect implementation).
- Bob reviews the patch, marks it "Accepted", "needs tests", and "patch needs improvement", and leaves a comment telling Alice how the patch could be improved.
- Alice updates the patch, adding tests (but not changing the implementation). She removes the two flags.
- Charlie reviews the patch and resets the "patch needs improvement" flag with another comment about improving the implementation.
- Alice updates the patch, fixing the implementation. She removes the "patch needs improvement" flag.
- Daisy reviews the patch, and marks it RFC.
- Jacob, a core developer, reviews the RFC patch, applies it to his checkout, and commits it.

Some tickets require much less feedback than this, but then again some tickets require much much more.

Triage stages

Below we describe in more detail the various stages that a ticket may flow through during its lifetime.

Unreviewed

The ticket has not been reviewed by anyone who felt qualified to make a judgment about whether the ticket contained a valid issue, a viable feature, or ought to be closed for any of the various reasons.

Accepted

The big grey area! The absolute meaning of "accepted" is that the issue described in the ticket is valid and is in some stage of being worked on. Beyond that there are several considerations:

- **Accepted + No Flags**

  The ticket is valid, but no one has submitted a patch for it yet. Often this means you could safely start writing a patch for it.

- **Accepted + Has Patch**

  The ticket is waiting for people to review the supplied patch. This means downloading the patch and trying it out, verifying that it contains tests and docs, running the test suite with the included patch, and leaving feedback on the ticket.

- **Accepted + Has Patch + (any other flag)**

  This means the ticket has been reviewed, and has been found to need further work. "Needs tests" and "Needs documentation" are self-explanatory. "Patch needs improvement" will generally be accompanied by a comment on the ticket explaining what is needed to improve the code.

Design Decision Needed

This stage is for issues which may be contentious, may be backwards incompatible, or otherwise involve high-level design decisions. These issues should be discussed either in the ticket comments or on django-developers.

If a ticket has been marked as "DDN", decisions are generally eventually made by the core committers, however that is not a requirement. See the *New contributors' FAQ* for "My ticket has been in DDN forever! What should I do?"

This stage will often be used for feature requests. It can also be used for issues that *might* be bugs, depending on opinion or interpretation. Obvious bugs (such as crashes, incorrect query results, or non-compliance with a standard) skip this stage and move straight to "Accepted".

Ready For Checkin

The ticket was reviewed by any member of the community other than the person who supplied the patch and found to meet all the requirements for a commit-ready patch. A core committer now needs to give the patch a final review prior to being committed. See the *New contributors' FAQ* for "My ticket has been in RFC forever! What should I do?"

Someday/Maybe

Generally only used for vague/high-level features or design ideas. These tickets are uncommon and overall less useful since they don't describe concrete actionable issues. They are enhancement requests that we might consider adding someday to the framework if an excellent patch is submitted. These tickets are not a high priority.

Fixed on a branch

Used to indicate that a ticket is resolved as part of a major body of work that will eventually be merged to trunk. Tickets in this stage generally don't need further work. This may happen in the case of major features/refactors in each release cycle, or as part of the annual Google Summer of Code efforts.

Other triage attributes

A number of flags, appearing as checkboxes in Trac, can be set on a ticket:

- **Has patch** This means the ticket has an associated *patch*. These will be reviewed to see if the patch is "good".
- **Needs documentation:** This flag is used for tickets with patches that need associated documentation. Complete documentation of features is a prerequisite before we can check them into the codebase.
- **Needs tests** This flags the patch as needing associated unit tests. Again, this is a required part of a valid patch.
- **Patch needs improvement** This flag means that although the ticket *has* a patch, it's not quite ready for checkin. This could mean the patch no longer applies cleanly, there is a flaw in the implementation, or that the code doesn't meet our standards.
- **Easy pickings** Tickets that would require small, easy, patches.

Tickets should be categorized by *type* between:

- **New Feature** For adding something new.
- **Bug** For when an existing thing is broken or not behaving as expected.
- **Cleanup/optimization** For when nothing is broken but something could be made cleaner, better, faster, stronger.

Tickets should also be classified into *components* indicating which area of the Django codebase they belong to. This makes tickets better organized and easier to find.

The *severity* attribute is used to identify blockers, that is, issues which should get fixed before releasing the next version of Django. Typically those issues are bugs causing regressions from earlier versions or potentially causing severe data losses. This attribute is quite rarely used and the vast majority of tickets have a severity of "Normal".

Finally, it is possible to use the *version* attribute to indicate in which version the reported bug was identified.

Closing Tickets

When a ticket has completed its useful lifecycle, it's time for it to be closed. Closing a ticket is a big responsibility, though. You have to be sure that the issue is really resolved, and you need to keep in mind that the reporter of the ticket may not be happy to have their ticket closed (unless it's fixed, of course). If you're not certain about closing a ticket, just leave a comment with your thoughts instead.

If you do close a ticket, you should always make sure of the following:

- Be certain that the issue is resolved.
- Leave a comment explaining the decision to close the ticket.
- If there is a way they can improve the ticket to reopen it, let them know.
- If the ticket is a duplicate, reference the original ticket. Also cross-reference the closed ticket by leaving a comment in the original one – this allows to access more related information about the reported bug or requested feature.

- **Be polite.** No one likes having their ticket closed. It can be frustrating or even discouraging. The best way to avoid turning people off from contributing to Django is to be polite and friendly and to offer suggestions for how they could improve this ticket and other tickets in the future.

A ticket can be resolved in a number of ways:

- **fixed** Used by the core developers once a patch has been rolled into Django and the issue is fixed.
- **invalid** Used if the ticket is found to be incorrect. This means that the issue in the ticket is actually the result of a user error, or describes a problem with something other than Django, or isn't a bug report or feature request at all (for example, some new users submit support queries as tickets).
- **wontfix** Used when a core developer decides that this request is not appropriate for consideration in Django. This is usually chosen after discussion in the django-developers mailing list. Feel free to start or join in discussions of "wontfix" tickets on the django-developers mailing list, but please do not reopen tickets closed as "wontfix" by a *core developer*.
- **duplicate** Used when another ticket covers the same issue. By closing duplicate tickets, we keep all the discussion in one place, which helps everyone.
- **worksforme** Used when the ticket doesn't contain enough detail to replicate the original bug.
- **needsinfo** Used when the ticket does not contain enough information to replicate the reported issue but is potentially still valid. The ticket should be reopened when more information is supplied.

If you believe that the ticket was closed in error – because you're still having the issue, or it's popped up somewhere else, or the triagers have made a mistake – please reopen the ticket and provide further information. Again, please do not reopen tickets that have been marked as "wontfix" by core developers and bring the issue to django-developers instead.

How can I help with triaging?

Although the core developers make the big decisions in the ticket triage process, there's a lot that general community members can do to help the triage process. Really, **ANYONE** can help.

Start by creating an account on Trac. If you have an account but have forgotten your password, you can reset it using the password reset page.

Then, you can help out by:

- Closing "Unreviewed" tickets as "invalid", "worksforme" or "duplicate."
- Promoting "Unreviewed" tickets to "Design decision needed" if a design decision needs to be made, or "Accepted" in case of obvious bugs or sensible, clearly defined, feature requests.
- Correcting the "Needs tests", "Needs documentation", or "Has patch" flags for tickets where they are incorrectly set.
- Setting the "Easy pickings" flag for tickets that are small and relatively straightforward.
- Checking that old tickets are still valid. If a ticket hasn't seen any activity in a long time, it's possible that the problem has been fixed but the ticket hasn't yet been closed.
- Contacting the owners of tickets that have been claimed but have not seen any recent activity. If the owner doesn't respond after a week or so, remove the owner's claim on the ticket.
- Identifying trends and themes in the tickets. If there a lot of bug reports about a particular part of Django, it may indicate we should consider refactoring that part of the code. If a trend is emerging, you should raise it for discussion (referencing the relevant tickets) on django-developers.
- Set the *type* of tickets that are still uncategorized.

- Verify if patches submitted by other users are correct. If they do and also contain appropriate documentation and tests then move them to the "Ready for Checkin" stage. If they don't then leave a comment to explain why and set the corresponding flags ("Patch needs improvement", "Needs tests" etc.).

Note: The Reports page contains links to many useful Trac queries, including several that are useful for triaging tickets and reviewing patches as suggested above.

You can also find more *Advice for new contributors*.

However, we do ask the following of all general community members working in the ticket database:

- Please **don't** close tickets as "wontfix." The core developers will make the final determination of the fate of a ticket, usually after consultation with the community.
- Please **don't** promote your own tickets to "Ready for checkin". You may mark other people's tickets which you've reviewed as "Ready for checkin", but you should get at minimum one other community member to review a patch that you submit.
- Please **don't** reverse a decision that has been made by a *core developer*. If you disagree with a decision that has been made, please post a message to django-developers.
- If you're unsure if you should be making a change, don't make the change but instead leave a comment with your concerns on the ticket, or post a message to django-developers. It's okay to be unsure, but your input is still valuable.

10.1.4 Writing code

So you'd like to write some code to improve Django. Awesome! Browse the following sections to find out how to give your code patches the best chances to be included in Django core:

Coding style

Please follow these coding standards when writing code for inclusion in Django.

Python style

- Unless otherwise specified, follow **PEP 8**.

  You could use a tool like pep8 to check for some problems in this area, but remember that **PEP 8** is only a guide, so respect the style of the surrounding code as a primary goal.

  One big exception to **PEP 8** is our preference of longer line lengths. We're well into the 21st Century, and we have high-resolution computer screens that can fit way more than 79 characters on a screen. Don't limit lines of code to 79 characters if it means the code looks significantly uglier or is harder to read.
- Use four spaces for indentation.
- Use underscores, not camelCase, for variable, function and method names (i.e. `poll.get_unique_voters()`, not `poll.getUniqueVoters`).
- Use `InitialCaps` for class names (or for factory functions that return classes).
- In docstrings, use "action words" such as:

```
def foo():
    """
    Calculates something and returns the result.
    """
    pass
```

Here's an example of what not to do:

```
def foo():
    """
    Calculate something and return the result.
    """
    pass
```

Template style

- In Django template code, put one (and only one) space between the curly brackets and the tag contents.

  Do this:

  ```
  {{ foo }}
  ```

  Don't do this:

  ```
  {{foo}}
  ```

View style

- In Django views, the first parameter in a view function should be called `request`.

  Do this:

  ```
  def my_view(request, foo):
      # ...
  ```

  Don't do this:

  ```
  def my_view(req, foo):
      # ...
  ```

Model style

- Field names should be all lowercase, using underscores instead of camelCase.

  Do this:

  ```
  class Person(models.Model):
      first_name = models.CharField(max_length=20)
      last_name = models.CharField(max_length=40)
  ```

  Don't do this:

  ```
  class Person(models.Model):
      FirstName = models.CharField(max_length=20)
      Last_Name = models.CharField(max_length=40)
  ```

- The `class Meta` should appear *after* the fields are defined, with a single blank line separating the fields and the class definition.

  Do this:

  ```
  class Person(models.Model):
      first_name = models.CharField(max_length=20)
      last_name = models.CharField(max_length=40)

      class Meta:
          verbose_name_plural = 'people'
  ```

  Don't do this:

  ```
  class Person(models.Model):
      first_name = models.CharField(max_length=20)
      last_name = models.CharField(max_length=40)
      class Meta:
          verbose_name_plural = 'people'
  ```

  Don't do this, either:

  ```
  class Person(models.Model):
      class Meta:
          verbose_name_plural = 'people'

      first_name = models.CharField(max_length=20)
      last_name = models.CharField(max_length=40)
  ```

- The order of model inner classes and standard methods should be as follows (noting that these are not all required):
  - All database fields
  - Custom manager attributes
  - `class Meta`
  - `def __unicode__()`
  - `def __str__()`
  - `def save()`
  - `def get_absolute_url()`
  - Any custom methods
- If `choices` is defined for a given model field, define the choices as a tuple of tuples, with an all-uppercase name, either near the top of the model module or just above the model class. Example:

  ```
  GENDER_CHOICES = (
      ('M', 'Male'),
      ('F', 'Female'),
  )
  ```

Use of `django.conf.settings`

Modules should not in general use settings stored in `django.conf.settings` at the top level (i.e. evaluated when the module is imported). The explanation for this is as follows:

Manual configuration of settings (i.e. not relying on the DJANGO_SETTINGS_MODULE environment variable) is allowed and possible as follows:

```
from django.conf import settings

settings.configure({}, SOME_SETTING='foo')
```

However, if any setting is accessed before the settings.configure line, this will not work. (Internally, settings is a LazyObject which configures itself automatically when the settings are accessed if it has not already been configured).

So, if there is a module containing some code as follows:

```
from django.conf import settings
from django.core.urlresolvers import get_callable

default_foo_view = get_callable(settings.FOO_VIEW)
```

...then importing this module will cause the settings object to be configured. That means that the ability for third parties to import the module at the top level is incompatible with the ability to configure the settings object manually, or makes it very difficult in some circumstances.

Instead of the above code, a level of laziness or indirection must be used, such as django.utils.functional.LazyObject, django.utils.functional.lazy() or lambda.

Miscellaneous

- Mark all strings for internationalization; see the *i18n documentation* for details.
- Remove import statements that are no longer used when you change code. The most common tools for this task are pyflakes and pylint.
- Systematically remove all trailing whitespaces from your code as those add unnecessary bytes, add visual clutter to the patches and can also occasionally cause unnecessary merge conflicts. Some IDE's can be configured to automatically remove them and most VCS tools can be set to highlight them in diff outputs. Note, however, that patches which only remove whitespace (or only make changes for nominal **PEP 8** conformance) are likely to be rejected, since they only introduce noise rather than code improvement. Tidy up when you're next changing code in the area.
- Please don't put your name in the code you contribute. Our policy is to keep contributors' names in the AUTHORS file distributed with Django – not scattered throughout the codebase itself. Feel free to include a change to the AUTHORS file in your patch if you make more than a single trivial change.

Unit tests

Django comes with a test suite of its own, in the tests directory of the code base. It's our policy to make sure all tests pass at all times.

The tests cover:

- Models and the database API (tests/modeltests),
- Everything else in core Django code (tests/regressiontests),
- *Contrib apps* (django/contrib/<app>/tests).

We appreciate any and all contributions to the test suite!

The Django tests all use the testing infrastructure that ships with Django for testing applications. See *Testing Django applications* for an explanation of how to write new tests.

Running the unit tests

Quickstart Running the tests requires a Django settings module that defines the databases to use. To make it easy to get started, Django provides a sample settings module that uses the SQLite database. To run the tests with this sample `settings` module, `cd` into the Django `tests/` directory and run:

```
./runtests.py --settings=test_sqlite
```

If you get an `ImportError: No module named django.contrib` error, you need to add your install of Django to your `PYTHONPATH`.

Using another `settings` module The included settings module allows you to run the test suite using SQLite. If you want to test behavior using a different database (and if you're proposing patches for Django, it's a good idea to test across databases), you may need to define your own settings file.

To run the tests with different settings, `cd` to the `tests/` directory and type:

```
./runtests.py --settings=path.to.django.settings
```

The `DATABASES` setting in this test settings module needs to define two databases:

- A `default` database. This database should use the backend that you want to use for primary testing
- A database with the alias `other`. The `other` database is used to establish that queries can be directed to different databases. As a result, this database can use any backend you want. It doesn't need to use the same backend as the `default` database (although it can use the same backend if you want to).

If you're using a backend that isn't SQLite, you will need to provide other details for each database:

- The `USER` option for each of your databases needs to specify an existing user account for the database.
- The `PASSWORD` option needs to provide the password for the `USER` that has been specified.
- The `NAME` option must be the name of an existing database to which the given user has permission to connect. The unit tests will not touch this database; the test runner creates a new database whose name is `NAME` prefixed with `test_`, and this test database is deleted when the tests are finished. This means your user account needs permission to execute `CREATE DATABASE`.

You will also need to ensure that your database uses UTF-8 as the default character set. If your database server doesn't use UTF-8 as a default charset, you will need to include a value for `TEST_CHARSET` in the settings dictionary for the applicable database.

Running only some of the tests Django's entire test suite takes a while to run, and running every single test could be redundant if, say, you just added a test to Django that you want to run quickly without running everything else. You can run a subset of the unit tests by appending the names of the test modules to `runtests.py` on the command line.

For example, if you'd like to run tests only for generic relations and internationalization, type:

```
./runtests.py --settings=path.to.settings generic_relations i18n
```

How do you find out the names of individual tests? Look in `tests/modeltests` and `tests/regressiontests` — each directory name there is the name of a test. Contrib app names are also valid test names.

If you just want to run a particular class of tests, you can specify a list of paths to individual test classes. For example, to run the `TranslationTests` of the `i18n` module, type:

```
./runtests.py --settings=path.to.settings i18n.TranslationTests
```

Going beyond that, you can specify an individual test method like this:

```
./runtests.py --settings=path.to.settings i18n.TranslationTests.test_lazy_objects
```

Running the Selenium tests Some admin tests require Selenium 2, Firefox and Python >= 2.6 to work via a real Web browser. To allow those tests to run and not be skipped, you must install the selenium package (version > 2.13) into your Python path.

Then, run the tests normally, for example:

```
./runtests.py --settings=test_sqlite admin_inlines
```

Running all the tests If you want to run the full suite of tests, you'll need to install a number of dependencies:

- PyYAML
- Markdown
- Textile
- Docutils
- setuptools
- memcached, plus a *supported Python binding*
- gettext (*gettext on Windows*)
- selenium (if also using Python >= 2.6)

If you want to test the memcached cache backend, you'll also need to define a `CACHES` setting that points at your memcached instance.

Each of these dependencies is optional. If you're missing any of them, the associated tests will be skipped.

Contrib apps

Tests for contrib apps go in their respective directories under `django/contrib`, in a `tests.py` file. You can split the tests over multiple modules by using a `tests` directory in the normal Python way.

For the tests to be found, a `models.py` file must exist, even if it's empty. If you have URLs that need to be mapped, put them in `tests/urls.py`.

To run tests for just one contrib app (e.g. `markup`), use the same method as above:

```
./runtests.py --settings=settings markup
```

Submitting patches

We're always grateful for patches to Django's code. Indeed, bug reports with associated patches will get fixed *far* more quickly than those without patches.

Typo fixes and trivial documentation changes

If you are fixing a really trivial issue, for example changing a word in the documentation, the preferred way to provide the patch is using GitHub pull requests without a Trac ticket. Trac tickets are still acceptable.

See the *Working with Git and GitHub* for more details on how to use pull requests.

"Claiming" tickets

In an open-source project with hundreds of contributors around the world, it's important to manage communication efficiently so that work doesn't get duplicated and contributors can be as effective as possible.

Hence, our policy is for contributors to "claim" tickets in order to let other developers know that a particular bug or feature is being worked on.

If you have identified a contribution you want to make and you're capable of fixing it (as measured by your coding ability, knowledge of Django internals and time availability), claim it by following these steps:

- Create an account to use in our ticket system. If you have an account but have forgotten your password, you can reset it using the password reset page.
- If a ticket for this issue doesn't exist yet, create one in our ticket tracker.
- If a ticket for this issue already exists, make sure nobody else has claimed it. To do this, look at the "Assigned to" section of the ticket. If it's assigned to "nobody," then it's available to be claimed. Otherwise, somebody else is working on this ticket, and you either find another bug/feature to work on, or contact the developer working on the ticket to offer your help.
- Log into your account, if you haven't already, by clicking "Login" in the upper right of the ticket page.
- Claim the ticket:
  1. click the "accept" radio button under "Action" near the bottom of the page,
  2. then click "Submit changes."

Ticket claimers' responsibility Once you've claimed a ticket, you have a responsibility to work on that ticket in a reasonably timely fashion. If you don't have time to work on it, either unclaim it or don't claim it in the first place!

If there's no sign of progress on a particular claimed ticket for a week or two, another developer may ask you to relinquish the ticket claim so that it's no longer monopolized and somebody else can claim it.

If you've claimed a ticket and it's taking a long time (days or weeks) to code, keep everybody updated by posting comments on the ticket. If you don't provide regular updates, and you don't respond to a request for a progress report, your claim on the ticket may be revoked.

As always, more communication is better than less communication!

Which tickets should be claimed? Of course, going through the steps of claiming tickets is overkill in some cases.

In the case of small changes, such as typos in the documentation or small bugs that will only take a few minutes to fix, you don't need to jump through the hoops of claiming tickets. Just submit your patch and be done with it.

Of course, it is *always* acceptable, regardless whether someone has claimed it or not, to submit patches to a ticket if you happen to have a patch ready.

Patch style

Make sure that any contribution you do fulfills at least the following requirements:

- The code required to fix a problem or add a feature is an essential part of a patch, but it is not the only part. A good patch should also include a *regression test* to validate the behavior that has been fixed and to prevent the problem from arising again. Also, if some tickets are relevant to the code that you've written, mention the ticket numbers in some comments in the test so that one can easily trace back the relevant discussions after your patch gets committed, and the tickets get closed.

- If the code associated with a patch adds a new feature, or modifies behavior of an existing feature, the patch should also contain documentation.

You can use either GitHub branches and pull requests or direct patches to publish your work. If you use the Git workflow, then you should announce your branch in the ticket by including a link to your branch. When you think your work is ready to be merged in create a pull request.

See the *Working with Git and GitHub* documentation for mode details.

You can also use patches in Trac. When using this style, follow these guidelines.

- Submit patches in the format returned by the `git diff` command. An exception is for code changes that are described more clearly in plain English than in code. Indentation is the most common example; it's hard to read patches when the only difference in code is that it's indented.
- Attach patches to a ticket in the ticket tracker, using the "attach file" button. Please *don't* put the patch in the ticket description or comment unless it's a single line patch.
- Name the patch file with a `.diff` extension; this will let the ticket tracker apply correct syntax highlighting, which is quite helpful.

Regardless of the way you submit your work, follow these steps.

- Make sure your code matches our *Coding style*.
- Check the "Has patch" box on the ticket details. This will make it obvious that the ticket includes a patch, and it will add the ticket to the list of tickets with patches.

Non-trivial patches

A "non-trivial" patch is one that is more than a simple bug fix. It's a patch that introduces Django functionality and makes some sort of design decision.

If you provide a non-trivial patch, include evidence that alternatives have been discussed on django-developers.

If you're not sure whether your patch should be considered non-trivial, just ask.

Javascript patches

Django's admin system leverages the jQuery framework to increase the capabilities of the admin interface. In conjunction, there is an emphasis on admin javascript performance and minimizing overall admin media file size. Serving compressed or "minified" versions of javascript files is considered best practice in this regard.

To that end, patches for javascript files should include both the original code for future development (e.g. `foo.js`), and a compressed version for production use (e.g. `foo.min.js`). Any links to the file in the codebase should point to the compressed version.

Compressing JavaScript To simplify the process of providing optimized javascript code, Django includes a handy script which should be used to create a "minified" version. This script is located at `django/contrib/admin/static/admin/js/compress.py`.

Behind the scenes, `compress.py` is a front-end for Google's Closure Compiler which is written in Java. However, the Closure Compiler library is not bundled with Django directly, so those wishing to contribute complete javascript patches will need to download and install the library independently.

The Closure Compiler library requires Java version 6 or higher (Java 1.6 or higher on Mac OS X. Note that Mac OS X 10.5 and earlier did not ship with Java 1.6 by default, so it may be necessary to upgrade your Java installation before

the tool will be functional. Also note that even after upgrading Java, the default `/usr/bin/java` command may remain linked to the previous Java binary, so relinking that command may be necessary as well.)

Please don't forget to run `compress.py` and include the `diff` of the minified scripts when submitting patches for Django's javascript.

Working with Git and GitHub

This section explains how the community can contribute code to Django via pull requests. If you're interested in how core developers handle them, see *Committing code*.

Below, we are going to show how to create a GitHub pull request containing the changes for Trac ticket #xxxxx. By creating a fully-ready pull request you will make the committers' job easier, meaning that your work is more likely to be merged into Django.

You could also upload a traditional patch to Trac, but it's less practical for reviews.

Installing Git

Django uses Git for its source control. You can download Git, but it's often easier to install with your operating system's package manager.

Django's Git repository is hosted on GitHub, and it is recommended that you also work using GitHub.

After installing Git the first thing you should do is setup your name and email:

```
$ git config --global user.name "Your Real Name"
$ git config --global user.email "you@email.com"
```

Note that `user.name` should be your real name, not your GitHub nick. GitHub should know the email you use in the `user.email` field, as this will be used to associate your commits with your GitHub account.

Setting up local repository

When you have created your GitHub account, with the nick "github_nick", and forked Django's repository, create a local copy of your fork:

```
git clone git@github.com:github_nick/django.git
```

This will create a new directory "django", containing a clone of your GitHub repository.

Your GitHub repository will be called "origin" in Git.

You should also setup django/django as an "upstream" remote (that is, tell git that the reference Django repository was the source of your fork of it):

```
git remote add upstream git@github.com:django/django.git
git fetch upstream
```

You can add other remotes similarly, for example:

```
git remote add akaariai git@github.com:akaariai/django.git
```

Working on a ticket

When working on a ticket create a new branch for the work, and base that work on upstream/master:

```
git checkout -b ticket_xxxxx upstream/master
```

The -b flag creates a new branch for you locally. Don't hesitate to create new branches even for the smallest things - that's what they are there for.

If instead you were working for a fix on the 1.4 branch, you would do:

```
git checkout -b ticket_xxxxx_1_4 upstream/stable/1.4.x
```

Assume the work is carried on ticket_xxxxx branch. Make some changes and commit them:

```
git commit
```

When writing the commit message, follow the *commit message guidelines* to ease the work of the committer. If you're uncomfortable with English, try at least to describe precisely what the commit does.

If you need to do additional work on your branch, commit as often as necessary:

```
git commit -m 'Added two more tests for edge cases'
```

Publishing work You can publish your work on GitHub just by doing:

```
git push origin ticket_xxxxx
```

When you go to your GitHub page you will notice a new branch has been created.

If you are working on a Trac ticket, you should mention in the ticket that your work is available from branch ticket_xxxxx of your github repo. Include a link to your branch.

Note that the above branch is called a "topic branch" in Git parlance. You are free to rewrite the history of this branch, by using `git rebase` for example. Other people shouldn't base their work on such a branch, because their clone would become corrupt when you edit commits.

There are also "public branches". These are branches other people are supposed to fork, so the history of these branches should never change. Good examples of public branches are the `master` and `stable/A.B.x` branches in the django/django repository.

When you think your work is ready to be pulled into Django, you should create a pull request at GitHub. A good pull request means:

- commits with one logical change in each, following the *coding style*,
- well-formed messages for each commit: a summary line and then paragraphs wrapped at 72 characters thereafter – see the *committing guidelines* for more details,
- documentation and tests, if needed – actually tests are always needed, except for documentation changes.

The test suite must pass and the documentation must build without warnings.

Once you have created your pull request, you should add a comment in the related Trac ticket explaining what you've done. In particular you should note the environment in which you ran the tests, for instance: "all tests pass under SQLite and MySQL".

Pull requests at GitHub have only two states: open and closed. The committer who will deal with your pull request has only two options: merge it or close it. For this reason, it isn't useful to make a pull request until the code is ready for merging – or sufficiently close that a committer will finish it himself.

Rebasing branches In the example above you created two commits, the "Fixed ticket_xxxxx" commit and "Added two more tests" commit.

We do not want to have the entire history of your working process in your repository. Your commit "Added two more tests" would be unhelpful noise. Instead, we would rather only have one commit containing all your work.

To rework the history of your branch you can squash the commits into one by using interactive rebase:

```
git rebase -i HEAD~2
```

The HEAD~2 above is shorthand for two latest commits. The above command will open an editor showing the two commits, prefixed with the word "pick".

Change the second line to "squash" instead. This will keep the first commit, and squash the second commit into the first one. Save and quit the editor. A second editor window should open, so you can reword the commit message for the commit now that it includes both your steps.

You can also use the "edit" option in rebase. This way you can change a single commit, for example to fix a typo in a docstring:

```
git rebase -i HEAD~3
# Choose edit, pick, pick for the commits
# Now you are able to rework the commit (use git add normally to add changes)
# When finished, commit work with "--amend" and continue
git commit --amend
# reword the commit message if needed
git rebase --continue
# The second and third commits should be applied.
```

If your topic branch is already published at GitHub, for example if you're making minor changes to take into account a review, you will need to force- push the changes:

```
git push -f origin ticket_xxxxx
```

Note that this will rewrite history of ticket_xxxxx - if you check the commit hashes before and after the operation at GitHub you will notice that the commit hashes do not match any more. This is acceptable, as the branch is merely a topic branch, and nobody should be basing their work on it.

After upstream has changed When upstream (django/django) has changed, you should rebase your work. To do this, use:

```
git fetch upstream
git rebase
```

The work is automatically rebased using the branch you forked on, in the example case using upstream/master.

The rebase command removes all your local commits temporarily, applies the upstream commits, and then applies your local commits again on the work.

If there are merge conflicts you will need to resolve them and then use `git rebase --continue`. At any point you can use `git rebase --abort` to return to the original state.

Note that you want to *rebase* on upstream, not *merge* the upstream.

The reason for this is that by rebasing, your commits will always be *on top of* the upstream's work, not *mixed in with* the changes in the upstream. This way your branch will contain only commits related to its topic, which makes squashing easier.

After review

It is unusual to get any non-trivial amount of code into core without changes requested by reviewers. In this case, it is often a good idea to add the changes as one incremental commit to your work. This allows the reviewer to easily check what changes you have done.

In this case, do the changes required by the reviewer. Commit as often as necessary. Before publishing the changes, rebase your work. If you added two commits, you would run:

```
git rebase -i HEAD~2
```

Squash the second commit into the first. Write a commit message along the lines of:

```
Made changes asked in review by <reviewer>

- Fixed whitespace errors in foobar
- Reworded the docstring of bar()
```

Finally push your work back to your GitHub repository. Since you didn't touch the public commits during the rebase, you should not need to force-push:

```
git push origin ticket_xxxxx
```

Your pull request should now contain the new commit too.

Note that the committer is likely to squash the review commit into the previous commit when committing the code.

Summary

- Work on GitHub if you can.
- Announce your work on the Trac ticket by linking to your GitHub branch.
- When you have something ready, make a pull request.
- Make your pull requests as good as you can.
- When doing fixes to your work, use `git rebase -i` to squash the commits.
- When upstream has changed, do `git fetch upstream; git rebase`.

10.1.5 Writing documentation

We place a high importance on consistency and readability of documentation. After all, Django was created in a journalism environment! So we treat our documentation like we treat our code: we aim to improve it as often as possible.

Documentation changes generally come in two forms:

- General improvements: typo corrections, error fixes and better explanations through clearer writing and more examples.
- New features: documentation of features that have been added to the framework since the last release.

This section explains how writers can craft their documentation changes in the most useful and least error-prone ways.

Getting the raw documentation

Though Django's documentation is intended to be read as HTML at https://docs.djangoproject.com/, we edit it as a collection of text files for maximum flexibility. These files live in the top-level `docs/` directory of a Django release.

If you'd like to start contributing to our docs, get the development version of Django from the source code repository (see *Installing the development version*). The development version has the latest-and-greatest documentation, just as it has latest-and-greatest code. Generally, we only revise documentation in the development version, as our policy is to freeze documentation for existing releases (see *Differences between versions*).

Getting started with Sphinx

Django's documentation uses the Sphinx documentation system, which in turn is based on docutils. The basic idea is that lightly-formatted plain-text documentation is transformed into HTML, PDF, and any other output format.

To actually build the documentation locally, you'll currently need to install Sphinx – `sudo pip install Sphinx` should do the trick.

Note: Building the Django documentation requires Sphinx 1.0.2 or newer. Sphinx also requires the Pygments library for syntax highlighting; building the Django documentation requires Pygments 1.1 or newer (a new-enough version should automatically be installed along with Sphinx).

Then, building the HTML is easy; just `make html` (or `make.bat html` on Windows) from the `docs` directory.

To get started contributing, you'll want to read the *reStructuredText Primer*. After that, you'll want to read about the *Sphinx-specific markup* that's used to manage metadata, indexing, and cross-references.

Commonly used terms

Here are some style guidelines on commonly used terms throughout the documentation:

- **Django** – when referring to the framework, capitalize Django. It is lowercase only in Python code and in the djangoproject.com logo.
- **email** – no hyphen.
- **MySQL**, **PostgreSQL**, **SQLite**
- **Python** – when referring to the language, capitalize Python.
- **realize**, **customize**, **initialize**, etc. – use the American "ize" suffix, not "ise."
- **subclass** – it's a single word without a hyphen, both as a verb ("subclass that model") and as a noun ("create a subclass").
- **Web**, **World Wide Web**, **the Web** – note Web is always capitalized when referring to the World Wide Web.
- **Web site** – use two words, with Web capitalized.

Django-specific terminology

- **model** – it's not capitalized.
- **template** – it's not capitalized.
- **URLconf** – use three capitalized letters, with no space before "conf."
- **view** – it's not capitalized.

Guidelines for reStructuredText files

These guidelines regulate the format of our reST (reStructuredText) documentation:

- In section titles, capitalize only initial words and proper nouns.
- Wrap the documentation at 80 characters wide, unless a code example is significantly less readable when split over two lines, or for another good reason.
- The main thing to keep in mind as you write and edit docs is that the more semantic markup you can add the better. So:

  ```
  Add ``django.contrib.auth`` to your ``INSTALLED_APPS``...
  ```

  Isn't nearly as helpful as:

  ```
  Add :mod:`django.contrib.auth` to your :setting:`INSTALLED_APPS`...
  ```

  This is because Sphinx will generate proper links for the latter, which greatly helps readers. There's basically no limit to the amount of useful markup you can add.
- Use `intersphinx` to reference Python's and Sphinx' documentation.

Django-specific markup

Besides the Sphinx built-in markup, Django's docs defines some extra description units:

- Settings:

  ```
  .. setting:: INSTALLED_APPS
  ```

  To link to a setting, use `:setting:`INSTALLED_APPS``.
- Template tags:

  ```
  .. templatetag:: regroup
  ```

  To link, use `:ttag:`regroup``.
- Template filters:

  ```
  .. templatefilter:: linebreaksbr
  ```

  To link, use `:tfilter:`linebreaksbr``.
- Field lookups (i.e. `Foo.objects.filter(bar__exact=whatever)`):

  ```
  .. fieldlookup:: exact
  ```

  To link, use `:lookup:`exact``.
- `django-admin` commands:

  ```
  .. django-admin:: syncdb
  ```

  To link, use `:djadmin:`syncdb``.
- `django-admin` command-line options:

  ```
  .. django-admin-option:: --traceback
  ```

  To link, use `:djadminopt:`--traceback``.

Documenting new features

Our policy for new features is:

> All documentation of new features should be written in a way that clearly designates the features are only available in the Django development version. Assume documentation readers are using the latest release, not the development version.

Our preferred way for marking new features is by prefacing the features' documentation with: "`.. versionadded:: X.Y`", followed by an optional one line comment and a mandatory blank line.

General improvements, or other changes to the APIs that should be emphasized should use the "`.. versionchanged:: X.Y`" directive (with the same format as the `versionadded` mentioned above.

An example

For a quick example of how it all fits together, consider this hypothetical example:

- First, the `ref/settings.txt` document could have an overall layout like this:

```
========
Settings
========

...

.. _available-settings:

Available settings
==================

...

.. _deprecated-settings:

Deprecated settings
===================

...
```

- Next, the `topics/settings.txt` document could contain something like this:

```
You can access a :ref:`listing of all available settings
<available-settings>`. For a list of deprecated settings see
:ref:`deprecated-settings`.

You can find both in the :doc:`settings reference document
</ref/settings>`.
```

  We use the Sphinx `doc` cross reference element when we want to link to another document as a whole and the `ref` element when we want to link to an arbitrary location in a document.

- Next, notice how the settings are annotated:

```
.. setting:: ADMIN_FOR

ADMIN_FOR
---------

Default: ``()`` (Empty tuple)
```

```
Used for admin-site settings modules, this should be a tuple of
settings modules (in the format ``'foo.bar.baz'``) for which this site
is an admin.

The admin site uses this in its automatically-introspected
documentation of models, views and template tags.
```

This marks up the following header as the "canonical" target for the setting `ADMIN_FOR` This means any time I talk about `ADMIN_FOR`, I can reference it using `:setting:`ADMIN_FOR``.

That's basically how everything fits together.

Improving the documentation

A few small improvements can be made to make the documentation read and look better:

- Most of the various `index.txt` documents have *very* short or even non-existent intro text. Each of those documents needs a good short intro the content below that point.
- The glossary is very perfunctory. It needs to be filled out.
- Add more metadata targets. Lots of places look like:

  ```
  ``File.close()``
  ~~~~~~~~~~~~~~~~
  ```

  ... these should be:

  ```
  .. method:: File.close()
  ```

  That is, use metadata instead of titles.
- Add more links – nearly everything that's an inline code literal right now can probably be turned into a xref.

  See the `literals_to_xrefs.py` file in `_ext` – it's a shell script to help do this work.

  This will probably be a continuing, never-ending project.
- Add info field lists where appropriate.
- Whenever possible, use links. So, use `:setting:`ADMIN_FOR`` instead of ``ADMIN_FOR``.
- Use directives where appropriate. Some directives (e.g. `.. setting::`) are prefix-style directives; they go *before* the unit they're describing. These are known as "crossref" directives. Others (e.g. `.. class::`) generate their own markup; these should go inside the section they're describing. These are called "description units".

  You can tell which are which by looking at in `_ext/djangodocs.py`; it registers roles as one of the other.
- Add `.. code-block:: <lang>` to literal blocks so that they get highlighted.
- When referring to classes/functions/modules, etc., you'll want to use the fully-qualified name of the target (`:class:`django.contrib.contenttypes.models.ContentType``).

  Since this doesn't look all that awesome in the output – it shows the entire path to the object – you can prefix the target with a `~` (that's a tilde) to get just the "last bit" of that path. So `:class:`~django.contrib.contenttypes.models.ContentType`` will just display a link with the title "ContentType".

10.1.6 Localizing Django

Various parts of Django, such as the admin site and validation error messages, are internationalized. This means they display differently depending on each user's language or country. For this, Django uses the same internationalization and localization infrastructure available to Django applications, described in the *i18n documentation*.

Translations

Translations are contributed by Django users worldwide. The translation work is coordinated at Transifex.

If you find an incorrect translation or want to discuss specific translations, go to the Django project page. If you would like to help out with translating or add a language that isn't yet translated, here's what to do:

- Join the Django i18n mailing list and introduce yourself.
- Make sure you read the notes about *Specialties of Django translation*.
- Signup at Transifex and visit the Django project page.
- On the Django project page, choose the language you want to work on, **or** – in case the language doesn't exist yet – request a new language team by clicking on the "Request language" link and selecting the appropriate language.
- Then, click the "Join this Team" button to become a member of this team. Every team has at least one coordinator who is responsible to review your membership request. You can of course also contact the team coordinator to clarify procedural problems and handle the actual translation process.
- Once you are a member of a team choose the translation resource you want to update on the team page. For example the "core" resource refers to the translation catalogue that contains all non-contrib translations. Each of the contrib apps also have a resource (prefixed with "contrib").

Note: For more information about how to use Transifex, read the Transifex User Guide.

Formats

You can also review `conf/locale/<locale>/formats.py`. This file describes the date, time and numbers formatting particularities of your locale. See *Format localization* for details.

The format files aren't managed by the use of Transifex. To change them, you must *create a patch* against the Django source tree, as for any code change:

- Create a diff against the current Subversion trunk.
- Open a ticket in Django's ticket system, set its `Component` field to `Translations`, and attach the patch to it.

10.1.7 Committing code

This section is addressed to the *Django committers* and to anyone interested in knowing how code gets committed into Django core. If you're a community member who wants to contribute code to Django, have a look at *Working with Git and GitHub* instead.

Commit access

Django has two types of committers:

Core committers These are people who have a long history of contributions to Django's codebase, a solid track record of being polite and helpful on the mailing lists, and a proven desire to dedicate serious time to Django's development. The bar is high for full commit access.

Partial committers These are people who are "domain experts." They have direct check-in access to the subsystems that fall under their jurisdiction, and they're given a formal vote in questions that involve their subsystems. This type of access is likely to be given to someone who contributes a large sub-framework to Django and wants to continue to maintain it.

Partial commit access is granted by the same process as full committers. However, the bar is set lower; proven expertise in the area in question is likely to be sufficient.

Decisions on new committers will follow the process explained in *How we make decisions*. To request commit access, please contact an existing committer privately. Public requests for commit access are potential flame-war starters, and will simply be ignored.

Handling pull requests

Since Django is now hosted at GitHub, many patches are provided in the form of pull requests.

When committing a pull request, make sure each individual commit matches the commit guidelines described below. Contributors are expected to provide the best pull requests possible. In practice however, committers - who will likely be more familiar with the commit guidelines - may decide to bring a commit up to standard themselves.

Here is one way to commit a pull request:

```
# Create a new branch tracking upstream/master -- upstream is assumed
# to be django/django.
git checkout -b pull_xxxxx upstream/master

# Download the patches from github and apply them.
curl https://github.com/django/django/pull/xxxxx.patch | git am
```

At this point, you can work on the code. Use `git rebase -i` and `git commit --amend` to make sure the commits have the expected level of quality. Once you're ready:

```
# Make sure master is ready to receive changes.
git checkout master
git pull upstream master
# Merge the work as "fast-forward" to master, to avoid a merge commit.
git merge --ff-only pull_xxxxx
# Check that only the changes you expect will be pushed to upstream.
git push --dry-run upstream master
# Push!
git push upstream master

# Get rid of the pull_xxxxx branch.
git branch -d pull_xxxxx
```

An alternative is to add the contributor's repository as a new remote, checkout the branch and work from there:

```
git remote add <contributor> https://github.com/<contributor>/django.git
git checkout pull_xxxxx <contributor> <contributor's pull request branch>
```

Yet another alternative is to fetch the branch without adding the contributor's repository as a remote:

```
git fetch https://github.com/<contributor>/django.git <contributor's pull request branch>
git checkout -b pull_xxxxx FETCH_HEAD
```

At this point, you can work on the code and continue as above.

GitHub provides a one-click merge functionality for pull requests. This should only be used if the pull request is 100% ready, and you have checked it for errors (or trust the request maker enough to skip checks). Currently, it isn't possible to check that the tests pass and that the docs build without downloading the changes to your development environment.

When rewriting the commit history of a pull request, the goal is to make Django's commit history as usable as possible:

- If a patch contains back-and-forth commits, then rewrite those into one. Typically, a commit can add some code, and a second commit can fix stylistic issues introduced in the first commit.
- Separate changes to different commits by logical grouping: if you do a stylistic cleanup at the same time as you do other changes to a file, separating the changes into two different commits will make reviewing history easier.
- Beware of merges of upstream branches in the pull requests.
- Tests should pass and docs should build after each commit. Neither the tests nor the docs should emit warnings.
- Trivial and small patches usually are best done in one commit. Medium to large work should be split into multiple commits if possible.

Practicality beats purity, so it is up to each committer to decide how much history mangling to do for a pull request. The main points are engaging the community, getting work done, and having a usable commit history.

Committing guidelines

In addition, please follow the following guidelines when committing code to Django's Git repository:

- Never change the published history of django/django branches! **Never force- push your changes to django/django.** If you absolutely must (for security reasons for example) first discuss the situation with the core team.
- For any medium-to-big changes, where "medium-to-big" is according to your judgment, please bring things up on the django-developers mailing list before making the change.

  If you bring something up on django-developers and nobody responds, please don't take that to mean your idea is great and should be implemented immediately because nobody contested it. Django's lead developers don't have a lot of time to read mailing-list discussions immediately, so you may have to wait a couple of days before getting a response.
- Write detailed commit messages in the past tense, not present tense.
  - Good: "Fixed Unicode bug in RSS API."
  - Bad: "Fixes Unicode bug in RSS API."
  - Bad: "Fixing Unicode bug in RSS API."

  The commit message should be in lines of 72 chars maximum. There should be a subject line, separated by a blank line and then paragraphs of 72 char lines. The limits are soft. For the subject line, shorter is better. In the body of the commit message more detail is better than less:

  ```
  Fixed #18307 -- Added git workflow guidelines

  Refactored the Django's documentation to remove mentions of SVN
  specific tasks. Added guidelines of how to use Git, GitHub, and
  how to use pull request together with Trac instead.
  ```

If the patch wasn't a pull request, you should credit the contributors in the commit message: "Thanks A for report, B for the patch and C for the review."

- For commits to a branch, prefix the commit message with the branch name. For example: "[1.4.x] Fixed #xxxxx – Added support for mind reading."
- Limit commits to the most granular change that makes sense. This means, use frequent small commits rather than infrequent large commits. For example, if implementing feature X requires a small change to library Y, first commit the change to library Y, then commit feature X in a separate commit. This goes a *long way* in helping all core Django developers follow your changes.
- Separate bug fixes from feature changes. Bugfixes may need to be backported to the stable branch, according to the *backwards-compatibility policy*.
- If your commit closes a ticket in the Django ticket tracker, begin your commit message with the text "Fixed #xxxxx", where "xxxxx" is the number of the ticket your commit fixes. Example: "Fixed #123 – Added whizbang feature.". We've rigged Trac so that any commit message in that format will automatically close the referenced ticket and post a comment to it with the full commit message.

  If your commit closes a ticket and is in a branch, use the branch name first, then the "Fixed #xxxxx." For example: "[1.4.x] Fixed #123 – Added whizbang feature."

  For the curious, we're using a Trac plugin for this.
- If your commit references a ticket in the Django ticket tracker but does *not* close the ticket, include the phrase "Refs #xxxxx", where "xxxxx" is the number of the ticket your commit references. This will automatically post a comment to the appropriate ticket.
- Write commit messages for backports using this pattern:

```
[<Django version>] Fixed <ticket> -- <description>

Backport of <revision> from <branch>.
```

  For example:

```
[1.3.x] Fixed #17028 - Changed diveintopython.org -> diveintopython.net.

Backport of 80c0cbf1c97047daed2c5b41b296bbc56fe1d7e3 from master.
```

Reverting commits

Nobody's perfect; mistakes will be committed.

But try very hard to ensure that mistakes don't happen. Just because we have a reversion policy doesn't relax your responsibility to aim for the highest quality possible. Really: double-check your work, or have it checked by another committer, **before** you commit it in the first place!

When a mistaken commit is discovered, please follow these guidelines:

- If possible, have the original author revert his/her own commit.
- Don't revert another author's changes without permission from the original author.
- Use git revert – this will make a reverse commit, but the original commit will still be part of the commit history.
- If the original author can't be reached (within a reasonable amount of time – a day or so) and the problem is severe – crashing bug, major test failures, etc – then ask for objections on the django-developers mailing list then revert if there are none.
- If the problem is small (a feature commit after feature freeze, say), wait it out.

- If there's a disagreement between the committer and the reverter-to-be then try to work it out on the django-developers mailing list. If an agreement can't be reached then it should be put to a vote.
- If the commit introduced a confirmed, disclosed security vulnerability then the commit may be reverted immediately without permission from anyone.
- The release branch maintainer may back out commits to the release branch without permission if the commit breaks the release branch.
- If you mistakenly push a topic branch to django/django, just delete it. For instance, if you did: `git push upstream feature_antigravity`, just do a reverse push: `git push upstream :feature_antigravity`.

10.2 Django committers

10.2.1 The original team

Django originally started at World Online, the Web department of the Lawrence Journal-World of Lawrence, Kansas, USA.

Adrian Holovaty Adrian is a Web developer with a background in journalism. He's known in journalism circles as one of the pioneers of "journalism via computer programming", and in technical circles as "the guy who invented Django."

He was lead developer at World Online for 2.5 years, during which time Django was developed and implemented on World Online's sites. He's now the leader and founder of EveryBlock, a "news feed for your block".

Adrian lives in Chicago, USA.

Simon Willison Simon is a well-respected Web developer from England. He had a one-year internship at World Online, during which time he and Adrian developed Django from scratch. The most enthusiastic Brit you'll ever meet, he's passionate about best practices in Web development and maintains a well-read web-development blog.

Simon lives in Brighton, England.

Jacob Kaplan-Moss Jacob is a partner at Revolution Systems which provides support services around Django and related open source technologies. A good deal of Jacob's work time is devoted to working on Django. Jacob previously worked at World Online, where Django was invented, where he was the lead developer of Ellington, a commercial Web publishing platform for media companies.

Jacob lives in Lawrence, Kansas, USA.

Wilson Miner Wilson's design-fu is what makes Django look so nice. He designed the Web site you're looking at right now, as well as Django's acclaimed admin interface. Wilson is the designer for EveryBlock.

Wilson lives in San Francisco, USA.

10.2.2 Current developers

Currently, Django is led by a team of volunteers from around the globe.

BDFLs

Adrian and Jacob are the Co-Benevolent Dictators for Life of Django. When "rough consensus and working code" fails, they're the ones who make the tough decisions.

Core developers

These are the folks who have a long history of contributions, a solid track record of being helpful on the mailing lists, and a proven desire to dedicate serious time to Django. In return, they've been granted the coveted commit bit, and have free rein to hack on all parts of Django.

Malcolm Tredinnick Malcolm originally wanted to be a mathematician, somehow ended up a software developer. He's contributed to many Open Source projects, has served on the board of the GNOME foundation, and will kick your ass at chess.

When he's not busy being an International Man of Mystery, Malcolm lives in Sydney, Australia.

Russell Keith-Magee Russell studied physics as an undergraduate, and studied neural networks for his PhD. His first job was with a startup in the defense industry developing simulation frameworks. Over time, mostly through work with Django, he's become more involved in Web development.

Russell has helped with several major aspects of Django, including a couple major internal refactorings, creation of the test system, and more.

Russell lives in the most isolated capital city in the world — Perth, Australia.

Joseph Kocherhans Joseph is currently a developer at EveryBlock, and previously worked for the Lawrence Journal-World where he built most of the backend for their Marketplace site. He often disappears for several days into the woods, attempts to teach himself computational linguistics, and annoys his neighbors with his Charango playing.

Joseph's first contribution to Django was a series of improvements to the authorization system leading up to support for pluggable authorization. Since then, he's worked on the new forms system, its use in the admin, and many other smaller improvements.

Joseph lives in Chicago, USA.

Luke Plant At University Luke studied physics and Materials Science and also met Michael Meeks who introduced him to Linux and Open Source, re-igniting an interest in programming. Since then he has contributed to a number of Open Source projects and worked professionally as a developer.

Luke has contributed many excellent improvements to Django, including database-level improvements, the CSRF middleware and many unit tests.

Luke currently works for a church in Bradford, UK, and part-time as a freelance developer.

Brian Rosner Brian is currently the tech lead at Eldarion managing and developing Django / **Pinax**_ based Web sites. He enjoys learning more about programming languages and system architectures and contributing to open source projects. Brian is the host of the Django Dose podcasts.

Brian helped immensely in getting Django's "newforms-admin" branch finished in time for Django 1.0; he's now a full committer, continuing to improve on the admin and forms system.

Brian lives in Denver, Colorado, USA.

Gary Wilson Gary starting contributing patches to Django in 2006 while developing Web applications for The University of Texas (UT). Since, he has made contributions to the email and forms systems, as well as many other improvements and code cleanups throughout the code base.

Gary is currently a developer and software engineering graduate student at UT, where his dedication to spreading the ways of Python and Django never ceases.

Gary lives in Austin, Texas, USA.

Justin Bronn Justin Bronn is a computer scientist and attorney specializing in legal topics related to intellectual property and spatial law.

In 2007, Justin began developing `django.contrib.gis` in a branch, a.k.a. GeoDjango, which was merged in time for Django 1.0. While implementing GeoDjango, Justin obtained a deep knowledge of Django's internals including the ORM, the admin, and Oracle support.

Justin lives in Houston, Texas.

Karen Tracey Karen has a background in distributed operating systems (graduate school), communications software (industry) and crossword puzzle construction (freelance). The last of these brought her to Django, in late 2006, when she set out to put a Web front-end on her crossword puzzle database. That done, she stuck around in the community answering questions, debugging problems, etc. – because coding puzzles are as much fun as word puzzles.

Karen lives in Apex, NC, USA.

Jannis Leidel Jannis graduated in media design from Bauhaus-University Weimar, is the author of a number of pluggable Django apps and likes to contribute to Open Source projects like virtualenv and pip.

He has worked on Django's auth, admin and staticfiles apps as well as the form, core, internationalization and test systems. He currently works as the lead engineer at Gidsy.

Jannis lives in Berlin, Germany.

James Tauber James is the lead developer of **Pinax_** and the CEO and founder of Eldarion. He has been doing open source software since 1993, Python since 1998 and Django since 2006. He serves on the board of the Python Software Foundation and is currently on a leave of absence from a PhD in linguistics.

James currently lives in Boston, MA, USA but originally hails from Perth, Western Australia where he attended the same high school as Russell Keith-Magee.

Alex Gaynor Alex is a software engineer working at Rdio. He found Django in 2007 and has been addicted ever since he found out you don't need to write out your forms by hand. He has a small obsession with compilers. He's contributed to the ORM, forms, admin, and other components of Django.

Alex lives in San Francisco, CA, USA.

Andrew Godwin Andrew is a freelance Python developer and tinkerer, and has been developing against Django since 2007. He graduated from Oxford University with a degree in Computer Science, and has become most well known in the Django community for his work on South, the schema migrations library.

Andrew lives in London, UK.

Carl Meyer Carl has been working with Django since 2007 (long enough to remember queryset-refactor, but not magic-removal), and works as a freelance developer with OddBird. He became a Django contributor by accident, because fixing bugs is more interesting than working around them.

Carl lives in Rapid City, SD, USA.

Ramiro Morales Ramiro has been reading Django source code and submitting patches since mid-2006 after researching for a Python Web tool with matching awesomeness and being pointed to it by an old ninja.

A software developer in the electronic transactions industry, he is a living proof of the fact that anyone with enough enthusiasm can contribute to Django, learning a lot and having fun in the process.

Ramiro lives in Córdoba, Argentina.

Chris Beaven Chris has been submitting patches and suggesting crazy ideas for Django since early 2006. An advocate for community involvement and a long-term triager, he is still often found answering questions in the #django IRC channel.

Chris lives in Napier, New Zealand (adding to the pool of Oceanic core developers). He works remotely as a developer for Lincoln Loop.

Honza Král Honza first discovered Django in 2006 and started using it right away, first for school and personal projects and later in his full time job. He contributed various patches and fixes mostly to the newforms library,

newforms admin and, through participation in the Google Summer of Code project, assisted in creating the *model validation* functionality.

He is currently working for Whiskey Media in San Francisco developing awesome sites running on pure Django.

Idan Gazit As a self-professed design geek, Idan was initially attracted to Django sometime between magic-removal and queryset-refactor. Formally trained as a software engineer, Idan straddles the worlds of design and code, jack of two trades and master of none. He is passionate about usability and finding novel ways to extract meaning from data, and is a longtime photographer.

Idan previously accepted freelance work under the Pixane imprint, but now splits his days between his startup, Skills, and beautifying all things Django and Python.

Paul McMillan Paul found Django in 2008 while looking for a more structured approach to web programming. He stuck around after figuring out that the developers of Django had already invented many of the wheels he needed. His passion for breaking (and then fixing) things led to his current role working to maintain and improve the security of Django.

Paul works in Berkeley, California as a web developer and security consultant.

Julien Phalip Julien has a background in software engineering and human-computer interaction. As a Web developer, he enjoys tinkering with the backend as much as designing and coding user interfaces. Julien discovered Django in 2007 while doing his PhD in Computing Sciences. Since then he has contributed patches to various components of the framework, in particular the admin. Julien was a co-founder of the Interaction Consortium. He now works at Odopod, a digital agency based in San Francisco, CA, USA.

Aymeric Augustin Aymeric is an engineer with a background in mathematics and computer science. He chose Django because he believes that software should be simple, explicit and tested. His perfectionist tendencies quickly led him to triage tickets and contribute patches.

Aymeric has a pragmatic approach to software engineering, can't live without a continuous integration server, and likes proving that Django is a good choice for enterprise software.

He works in a management consulting company in Paris, France.

Claude Paroz Claude is a former teacher who fell in love with free software at the beginning of the 21st century. He's now working as freelancer in Web development in his native Switzerland. He has found in Django a perfect match for his needs of a stable, clean, documented and well-maintained Web framework.

He's also helping the GNOME Translation Project as maintainer of the Django-based l10n.gnome.org.

Anssi Kääriäinen Anssi works as a developer at Finnish National Institute for Health and Welfare. He is also a computer science student at Aalto University. In his work he uses Django for developing internal business applications and sees Django as a great match for that use case.

Anssi is interested in developing the object relational mapper (ORM) and all related features. He's also a fan of benckmarking and he tries keep Django as fast as possible.

Florian Apolloner Florian is currently studying Physics at the Graz University of Technology. Soon after he started using Django he joined the Ubuntuusers webteam to work on *Inyoka*, the software powering the whole Ubuntusers site.

For the time beeing he lives in Graz, Austria (not Australia ;)).

Specialists

James Bennett James is Django's release manager, and also contributes to the documentation and provide the occasional bugfix.

James came to Web development from philosophy when he discovered that programmers get to argue just as much while collecting much better pay. He lives in Lawrence, Kansas and previously worked at World Online; currently, he's part of the Web development team at Mozilla.

He keeps a blog, and enjoys fine port and talking to his car.

Ian Kelly Ian is responsible for Django's support for Oracle.

Matt Boersma Matt is also responsible for Django's Oracle support.

Jeremy Dunck Jeremy is the lead developer of Pegasus News, a personalized local site based in Dallas, Texas. An early contributor to Greasemonkey and Django, he sees technology as a tool for communication and access to knowledge.

Jeremy helped kick off GeoDjango development, and is mostly responsible for the serious speed improvements that signals received in Django 1.0.

Jeremy lives in Dallas, Texas, USA.

Simon Meers Simon discovered Django 0.96 during his Computer Science PhD research and has been developing with it full-time ever since. His core code contributions are mostly in Django's admin application. He is also helping to improve Django's documentation.

Simon works as a freelance developer based in Wollongong, Australia.

Gabriel Hurley Gabriel has been working with Django since 2008, shortly after the 1.0 release. Convinced by his business partner that Python and Django were the right direction for the company, he couldn't have been more happy with the decision. His contributions range across many areas in Django, but years of copy-editing and an eye for detail lead him to be particularly at home while working on Django's documentation.

Gabriel works as a web developer in Berkeley, CA, USA.

Tim Graham When exploring Web frameworks for an independent study project in the fall of 2008, Tim discovered Django and was lured to it by the documentation. He enjoys contributing to the docs because they're awesome.

Tim works as a software engineer and lives in Philadelphia, PA, USA.

10.2.3 Developers Emeritus

Georg "Hugo" Bauer Georg created Django's internationalization system, managed i18n contributions and made a ton of excellent tweaks, feature additions and bug fixes.

Robert Wittams Robert was responsible for the *first* refactoring of Django's admin application to allow for easier reuse and has made a ton of excellent tweaks, feature additions and bug fixes.

10.3 Django's release process

10.3.1 Official releases

Since version 1.0, Django's release numbering works as follows:

- Versions are numbered in the form `A.B` or `A.B.C`.
- `A` is the *major version* number, which is only incremented for major changes to Django, and these changes are not necessarily backwards-compatible. That is, code you wrote for Django 1.2 may break when we release Django 2.0.
- `B` is the *minor version* number, which is incremented for large yet backwards compatible changes. Code written for Django 1.2 will continue to work under Django 1.3. Exceptions to this rule will be listed in the release notes.

- `C` is the *micro version* number, which is incremented for bug and security fixes. A new micro-release will be 100% backwards-compatible with the previous micro-release. The only exception is when a security issue can't be fixed without breaking backwards-compatibility. If this happens, the release notes will provide detailed upgrade instructions.
- In some cases, we'll make alpha, beta, or release candidate releases. These are of the form `A.B alpha/beta/rc N`, which means the `Nth` alpha/beta/release candidate of version `A.B`.

In Subversion, each Django release will be tagged under `tags/releases`. If it's necessary to release a bug fix release or a security release that doesn't come from the trunk, we'll copy that tag to `branches/releases` to make the bug fix release.

Major releases

Major releases (1.0, 2.0, etc.) will happen very infrequently (think "years", not "months"), and will probably represent major, sweeping changes to Django.

Minor releases

Minor release (1.1, 1.2, etc.) will happen roughly every nine months – see release process, below for details. These releases will contain new features, improvements to existing features, and such. A minor release may deprecate certain features from previous releases. If a feature in version `A.B` is deprecated, it will continue to work in version `A.B+1`. In version `A.B+2`, use of the feature will raise a `DeprecationWarning` but will continue to work. Version `A.B+3` will remove the feature entirely.

So, for example, if we decided to remove a function that existed in Django 1.0:

- Django 1.1 will contain a backwards-compatible replica of the function which will raise a `PendingDeprecationWarning`. This warning is silent by default; you need to explicitly turn on display of these warnings.
- Django 1.2 will contain the backwards-compatible replica, but the warning will be promoted to a full-fledged `DeprecationWarning`. This warning is *loud* by default, and will likely be quite annoying.
- Django 1.3 will remove the feature outright.

Micro releases

Micro releases (1.0.1, 1.0.2, 1.1.1, etc.) will be issued at least once half-way between minor releases, and probably more often as needed.

These releases will be 100% compatible with the associated minor release, unless this is impossible for security reasons. So the answer to "should I upgrade to the latest micro release?" will always be "yes."

Each minor release of Django will have a "release maintainer" appointed. This person will be responsible for making sure that bug fixes are applied to both trunk and the maintained micro-release branch. This person will also work with the release manager to decide when to release the micro releases.

10.3.2 Supported versions

At any moment in time, Django's developer team will support a set of releases to varying levels:

- The current development trunk will get new features and bug fixes requiring major refactoring.
- Patches applied to the trunk will also be applied to the last minor release, to be released as the next micro release, when they fix critical problems:

  - Security issues.
  - Data-loss bugs.
  - Crashing bugs.
  - Major functionality bugs in newly-introduced features.

  The rule of thumb is that fixes will be backported to the last minor release for bugs that would have prevented a release in the first place.

- Security fixes will be applied to the current trunk and the previous two minor releases.
- Documentation fixes generally will be more freely backported to the last release branch, at the discretion of the committer, and they don't need to meet the "critical fixes only" bar. That's because it's highly advantageous to have the docs for the last release be up-to-date and correct, and the downside of backporting (risk of introducing regressions) is much less of a concern.

As a concrete example, consider a moment in time halfway between the release of Django 1.3 and 1.4. At this point in time:

- Features will be added to development trunk, to be released as Django 1.4.
- Critical bug fixes will be applied to a `1.3.X` branch, and released as 1.3.1, 1.3.2, etc.
- Security fixes will be applied to trunk, a `1.3.X` branch and a `1.2.X` branch. They will trigger the release of `1.3.1`, `1.2.1`, etc.
- Documentation fixes will be applied to trunk, and, if easily backported, to the `1.3.X` branch.

10.3.3 Release process

Django uses a time-based release schedule, with minor (i.e. 1.1, 1.2, etc.) releases every nine months, or more, depending on features.

After each release, and after a suitable cooling-off period of a few weeks, the core development team will examine the landscape and announce a timeline for the next release. Most releases will be scheduled in the 6-9 month range, but if we have bigger features to development we might schedule a longer period to allow for more ambitious work.

Release cycle

Each release cycle will be split into three periods, each lasting roughly one-third of the cycle:

Phase one: feature proposal

The first phase of the release process will be devoted to figuring out what features to include in the next version. This should include a good deal of preliminary work on those features – working code trumps grand design.

At the end of part one, the core developers will propose a feature list for the upcoming release. This will be broken into:

- "Must-have": critical features that will delay the release if not finished
- "Maybe" features: that will be pushed to the next release if not finished
- "Not going to happen": features explicitly deferred to a later release.

Anything that hasn't got at least some work done by the end of the first third isn't eligible for the next release; a design alone isn't sufficient.

Phase two: development

The second third of the release schedule is the "heads-down" working period. Using the roadmap produced at the end of phase one, we'll all work very hard to get everything on it done.

Longer release schedules will likely spend more than a third of the time in this phase.

At the end of phase two, any unfinished "maybe" features will be postponed until the next release. Though it shouldn't happen, any "must-have" features will extend phase two, and thus postpone the final release.

Phase two will culminate with an alpha release.

Phase three: bugfixes

The last third of a release is spent fixing bugs – no new features will be accepted during this time. We'll release a beta release about halfway through, and an rc complete with string freeze two weeks before the end of the schedule.

Bug-fix releases

After a minor release (e.g. 1.1), the previous release will go into bugfix mode.

A branch will be created of the form `branches/releases/1.0.X` to track bugfixes to the previous release. Critical bugs fixed on trunk must *also* be fixed on the bugfix branch; this means that commits need to cleanly separate bug fixes from feature additions. The developer who commits a fix to trunk will be responsible for also applying the fix to the current bugfix branch. Each bugfix branch will have a maintainer who will work with the committers to keep them honest on backporting bug fixes.

How this all fits together

Let's look at a hypothetical example for how this all first together. Imagine, if you will, a point about halfway between 1.1 and 1.2. At this point, development will be happening in a bunch of places:

- On trunk, development towards 1.2 proceeds with small additions, bugs fixes, etc. being checked in daily.
- On the branch "branches/releases/1.1.X", fixes for critical bugs found in the 1.1 release are checked in as needed. At some point, this branch will be released as "1.1.1", "1.1.2", etc.
- On the branch "branches/releases/1.0.X", security fixes are made if needed and released as "1.0.2", "1.0.3", etc.
- On feature branches, development of major features is done. These branches will be merged into trunk before the end of phase two.

10.4 Django Deprecation Timeline

This document outlines when various pieces of Django will be removed or altered in a backward incompatible way, following their deprecation, as per the *deprecation policy*. More details about each item can often be found in the release notes of two versions prior.

10.4.1 1.4

See the *Django 1.2 release notes* for more details on these changes.

- `CsrfResponseMiddleware` and `CsrfMiddleware` will be removed. Use the {% csrf_token %} template tag inside forms to enable CSRF protection. `CsrfViewMiddleware` remains and is enabled by default.
- The old imports for CSRF functionality (`django.contrib.csrf.*`), which moved to core in 1.2, will be removed.
- The `django.contrib.gis.db.backend` module will be removed in favor of the specific backends.
- `SMTPConnection` will be removed in favor of a generic E-mail backend API.
- The many to many SQL generation functions on the database backends will be removed.
- The ability to use the `DATABASE_*` family of top-level settings to define database connections will be removed.
- The ability to use shorthand notation to specify a database backend (i.e., `sqlite3` instead of `django.db.backends.sqlite3`) will be removed.
- The `get_db_prep_save`, `get_db_prep_value` and `get_db_prep_lookup` methods will have to support multiple databases.
- The `Message` model (in `django.contrib.auth`), its related manager in the `User` model (`user.message_set`), and the associated methods (`user.message_set.create()` and `user.get_and_delete_messages()`), will be removed. The *messages framework* should be used instead. The related `messages` variable returned by the auth context processor will also be removed. Note that this means that the admin application will depend on the messages context processor.
- Authentication backends will need to support the `obj` parameter for permission checking. The `supports_object_permissions` attribute will no longer be checked and can be removed from custom backends.
- Authentication backends will need to support the `AnonymousUser` class being passed to all methods dealing with permissions. The `supports_anonymous_user` variable will no longer be checked and can be removed from custom backends.
- The ability to specify a callable template loader rather than a `Loader` class will be removed, as will the `load_template_source` functions that are included with the built in template loaders for backwards compatibility.
- `django.utils.translation.get_date_formats()` and `django.utils.translation.get_partial_dat` These functions will be removed; use the locale-aware `django.utils.formats.get_format()` to get the appropriate formats.
- In `django.forms.fields`, the constants: `DEFAULT_DATE_INPUT_FORMATS`, `DEFAULT_TIME_INPUT_FORMATS` and `DEFAULT_DATETIME_INPUT_FORMATS` will be removed. Use `django.utils.formats.get_format()` to get the appropriate formats.
- The ability to use a function-based test runners will be removed, along with the `django.test.simple.run_tests()` test runner.
- The `views.feed()` view and `feeds.Feed` class in `django.contrib.syndication` will be removed. The class-based view `views.Feed` should be used instead.
- `django.core.context_processors.auth`. This release will remove the old method in favor of the new method in `django.contrib.auth.context_processors.auth`.
- The `postgresql` database backend will be removed, use the `postgresql_psycopg2` backend instead.
- The `no` language code will be removed and has been replaced by the `nb` language code.
- Authentication backends will need to define the boolean attribute `supports_inactive_user` until version 1.5 when it will be assumed that all backends will handle inactive users.

- `django.db.models.fields.XMLField` will be removed. This was deprecated as part of the 1.3 release. An accelerated deprecation schedule has been used because the field hasn't performed any role beyond that of a simple `TextField` since the removal of oldforms. All uses of `XMLField` can be replaced with `TextField`.
- The undocumented `mixin` parameter to the `open()` method of `django.core.files.storage.Storage` (and subclasses) will be removed.

10.4.2 1.5

See the *Django 1.3 release notes* for more details on these changes.

- Starting Django without a `SECRET_KEY` will result in an exception rather than a *DeprecationWarning*. (This is accelerated from the usual deprecation path; see the *Django 1.4 release notes*.)
- The `mod_python` request handler will be removed. The `mod_wsgi` handler should be used instead.
- The `template` attribute on `Response` objects returned by the *test client* will be removed. The `templates` attribute should be used instead.
- The `DjangoTestRunner` will be removed. Instead use a unittest-native class. The features of the `django.test.simple.DjangoTestRunner` (including fail-fast and Ctrl-C test termination) can currently be provided by the unittest-native `TextTestRunner`.
- The undocumented function `django.contrib.formtools.utils.security_hash()` will be removed, instead use `django.contrib.formtools.utils.form_hmac()`
- The function-based generic view modules will be removed in favor of their class-based equivalents, outlined *here*:
- The `AdminMediaHandler` will be removed. In its place use `StaticFilesHandler`.
- The `url` and `ssi` template tags will be modified so that the first argument to each tag is a template variable, not an implied string. Until then, the new-style behavior is provided in the `future` template tag library.
- The `reset` and `sqlreset` management commands will be removed.
- Authentication backends will need to support an inactive user being passed to all methods dealing with permissions. The `supports_inactive_user` attribute will no longer be checked and can be removed from custom backends.
- `transform()` will raise a `GEOSException` when called on a geometry with no SRID value.
- `CompatCookie` will be removed in favor of `SimpleCookie`.
- `django.core.context_processors.PermWrapper` and `django.core.context_processors.PermLookup` will be removed in favor of the corresponding `django.contrib.auth.context_processors.PermWrapper` and `django.contrib.auth.context_processors.PermLookupDict`, respectively.
- The `MEDIA_URL` or `STATIC_URL` settings will be required to end with a trailing slash to ensure there is a consistent way to combine paths in templates.
- `django.db.models.fields.URLField.verify_exists` will be removed. The feature was deprecated in 1.3.1 due to intractable security and performance issues and will follow a slightly accelerated deprecation timeframe.
- Translations located under the so-called *project path* will be ignored during the translation building process performed at runtime. The `LOCALE_PATHS` setting can be used for the same task by including the filesystem path to a `locale` directory containing non-app-specific translations in its value.
- The Markup contrib app will no longer support versions of Python-Markdown library earlier than 2.1. An accelerated timeline was used as this was a security related deprecation.

- The CACHE_BACKEND setting will be removed. The cache backend(s) should be specified in the CACHES setting.

10.4.3 1.6

See the *Django 1.4 release notes* for more details on these changes.

- The compatibility modules django.utils.copycompat and django.utils.hashcompat as well as the functions django.utils.itercompat.all and django.utils.itercompat.any will be removed. The Python builtin versions should be used instead.
- The csrf_response_exempt() and csrf_view_exempt() decorators will be removed. Since 1.4 csrf_response_exempt has been a no-op (it returns the same function), and csrf_view_exempt has been a synonym for django.views.decorators.csrf.csrf_exempt, which should be used to replace it.
- The CacheClass backend was split into two in Django 1.3 in order to introduce support for PyLibMC. The historical CacheClass will be removed in favor of MemcachedCache.
- The UK-prefixed objects of django.contrib.localflavor.uk will only be accessible through their GB-prefixed names (GB is the correct ISO 3166 code for United Kingdom).
- The IGNORABLE_404_STARTS and IGNORABLE_404_ENDS settings have been superseded by IGNORABLE_404_URLS in the 1.4 release. They will be removed.
- The *form wizard* has been refactored to use class-based views with pluggable backends in 1.4. The previous implementation will be removed.
- Legacy ways of calling cache_page() will be removed.
- The backward-compatibility shim to automatically add a debug-false filter to the 'mail_admins' logging handler will be removed. The LOGGING setting should include this filter explicitly if it is desired.
- The template tag django.contrib.admin.templatetags.adminmedia.admin_media_prefix() will be removed in favor of the generic static files handling.
- The builtin truncation functions django.utils.text.truncate_words() and django.utils.text.truncate_html_words() will be removed in favor of the django.utils.text.Truncator class.
- The GeoIP class was moved to django.contrib.gis.geoip in 1.4 – the shortcut in django.contrib.gis.utils will be removed.
- django.conf.urls.defaults will be removed. The functions include(), patterns() and url() plus handler404, handler500, are now available through django.conf.urls .
- The Databrowse contrib module will be removed.
- The functions setup_environ() and execute_manager() will be removed from django.core.management. This also means that the old (pre-1.4) style of manage.py file will no longer work.
- Setting the is_safe and needs_autoescape flags as attributes of template filter functions will no longer be supported.
- The attribute HttpRequest.raw_post_data was renamed to HttpRequest.body in 1.4. The backward compatibility will be removed – HttpRequest.raw_post_data will no longer work.

10.4.4 1.7

See the *Django 1.5 release notes* for more details on these changes.

- The module `django.utils.simplejson` will be removed. The standard library provides `json` which should be used instead.
- The function `django.utils.itercompat.product` will be removed. The Python builtin version should be used instead.
- Auto-correction of INSTALLED_APPS and TEMPLATE_DIRS settings when they are specified as a plain string instead of a tuple will be removed and raise an exception.
- The `mimetype` argument to `HttpResponse` `__init__` will be removed (`content_type` should be used instead).

10.4.5 2.0

- `django.views.defaults.shortcut()`. This function has been moved to `django.contrib.contenttypes.views.shortcut()` as part of the goal of removing all `django.contrib` references from the core Django codebase. The old shortcut will be removed in the 2.0 release.
- `ssi` and `url` template tags will be removed from the `future` template tag library (used during the 1.3/1.4 deprecation period).

10.5 The Django source code repository

When deploying a Django application into a real production environment, you will almost always want to use an official packaged release of Django.

However, if you'd like to try out in-development code from an upcoming release or contribute to the development of Django, you'll need to obtain a clone of Django's source code repository.

This document covers the way the code repository is laid out and how to work with and find things in it.

10.5.1 High-level overview

The Django source code repository uses Git to track changes to the code over time, so you'll need a copy of the Git client (a program called `git`) on your computer, and you'll want to familiarize yourself with the basics of how Git works.

Git's web site offers downloads for various operating systems. The site also contains vast amounts of documentation.

The Django Git repository is located online at github.com/django/django. It contains the full source code for all Django releases, which you can browse online.

The Git repository includes several branches:

- `master` contains the main in-development code which will become the next packaged release of Django. This is where most development activity is focused.
- `stable/A.B.x` are the maintenance branches. They are used to support older versions of Django.
- `soc20XX/<project>` branches were used by students who worked on Django during the 2009 and 2010 Google Summer of Code programs.

- `attic/<project>` branches were used to develop major or experimental new features without affecting the rest of Django's code.

The Git repository also contains tags. These are the exact revisions from which packaged Django releases were produced, since version 1.0.

The source code for the Djangoproject.com web site can be found at github.com/django/djangoproject.com.

10.5.2 The master branch

If you'd like to try out the in-development code for the next release of Django, or if you'd like to contribute to Django by fixing bugs or developing new features, you'll want to get the code from the master branch.

Note that this will get *all* of Django: in addition to the top-level `django` module containing Python code, you'll also get a copy of Django's documentation, test suite, packaging scripts and other miscellaneous bits. Django's code will be present in your clone as a directory named `django`.

To try out the in-development code with your own applications, simply place the directory containing your clone on your Python import path. Then `import` statements which look for Django will find the `django` module within your clone.

If you're going to be working on Django's code (say, to fix a bug or develop a new feature), you can probably stop reading here and move over to *the documentation for contributing to Django*, which covers things like the preferred coding style and how to generate and submit a patch.

10.5.3 Other branches

Django uses branches for two main purposes:

1. Development of major or experimental features, to keep them from affecting progress on other work in master.
2. Security and bugfix support for older releases of Django, during their support lifetimes.

Feature-development branches

Historical information

Since Django moved to Git in 2012, anyone can clone the repository and create his own branches, alleviating the need for official branches in the source code repository.

The following section is mostly useful if you're exploring the repository's history, for example if you're trying to understand how some features were designed.

Feature-development branches tend by their nature to be temporary. Some produce successful features which are merged back into Django's master to become part of an official release, but others do not; in either case there comes a time when the branch is no longer being actively worked on by any developer. At this point the branch is considered closed.

Unfortunately, Django used to be maintained with the Subversion revision control system, that has no standard way of indicating this. As a workaround, branches of Django which are closed and no longer maintained were moved into `attic`.

For reference, the following are branches whose code eventually became part of Django itself, and so are no longer separately maintained:

- `boulder-oracle-sprint`: Added support for Oracle databases to Django's object-relational mapper. This has been part of Django since the 1.0 release.
- `gis`: Added support for geographic/spatial queries to Django's object-relational mapper. This has been part of Django since the 1.0 release, as the bundled application `django.contrib.gis`.
- `i18n`: Added *internationalization support* to Django. This has been part of Django since the 0.90 release.
- `magic-removal`: A major refactoring of both the internals and public APIs of Django's object-relational mapper. This has been part of Django since the 0.95 release.
- `multi-auth`: A refactoring of *Django's bundled authentication framework* which added support for *authentication backends*. This has been part of Django since the 0.95 release.
- `new-admin`: A refactoring of *Django's bundled administrative application*. This became part of Django as of the 0.91 release, but was superseded by another refactoring (see next listing) prior to the Django 1.0 release.
- `newforms-admin`: The second refactoring of Django's bundled administrative application. This became part of Django as of the 1.0 release, and is the basis of the current incarnation of `django.contrib.admin`.
- `queryset-refactor`: A refactoring of the internals of Django's object-relational mapper. This became part of Django as of the 1.0 release.
- `unicode`: A refactoring of Django's internals to consistently use Unicode-based strings in most places within Django and Django applications. This became part of Django as of the 1.0 release.

When Django moved from SVN to Git, the information about branch merges wasn't preserved in the source code repository. This means that the `master` branch of Django doesn't contain merge commits for the above branches.

However, this information is available as a grafts file. You can restore it by putting the following lines in `.git/info/grafts` in your local clone:

```
ac64e91a0cadc57f4bc5cd5d66955832320ca7a1 553a20075e6991e7a60baee51ea68c8adc520d9a 0cb8e31823b2e9f05c
79e68c225b926302ebb29c808dda8afa49856f5c d0f57e7c7385a112cb9e19d314352fc5ed5b0747 aa239e3e5405933af6
5cf8f684237ab5addaf3549b2347c3adf107c0a7 cb45fd0ae20597306cd1f877efc99d9bd7cbee98 e27211a0deae2f1d40
f69cf70ed813a8cd7e1f963a14ae39103e8d5265 d5dbeaa9be359a4c794885c2e9f1b5a7e5e51fb8 d2fcbcf9d76d5bb8a6
aab3a418ac9293bb4abd7670f65d930cb0426d58 4ea7a11659b8a0ab07b0d2e847975f7324664f10 adf4b9311d5d64a2bd
ff60c5f9de3e8690d1e86f3e9e3f7248a15397c8 7ef212af149540aa2da577a960d0d87029fd1514 45b4288bb66a3cda40
9dda4abee1225db7a7b195b84c915fdd141a7260 4fe5c9b7ee09dc25921918a6dbb7605edb374bc9 3a7c14b583621272d4
a19ed8aea395e8e07164ff7d85bd7dff2f24edca dc375fb0f3b7fbae740e8cfcd791b8bccb8a4e66 42ea7a5ce8aece67d1
9c52d56f6f8a9cdafb231adf9f4110473099c9b5 c91a30f00fd182faf8ca5c03cd7dbcf8b735b458 4a5c5c78f2ecd4ed88
953badbea5a04159adbfa970f5805c0232b6a401 4c958b15b250866b70ded7d82aa532f1e57f96ae 5664a678b29ab04cad
471596fc1afcb9c6258d317c619eaf5fd394e797 4e89105d64bb9e04c409139a41e9c7aac263df4c 3e9035a9625c8a8a5e
9233d0426537615e06b78d28010d17d5a66adf44 6632739e94c6c38b4c5a86cf5c80c48ae50ac49f 18e151bc3f8a85f276
```

Additionally, the following branches are closed, but their code was never merged into Django and the features they aimed to implement were never finished:

- `full-history`
- `generic-auth`
- `multiple-db-support`
- `per-object-permissions`
- `schema-evolution`
- `schema-evolution-ng`
- `search-api`
- `sqlalchemy`

All of the above-mentioned branches now reside in `attic`.

Finally, the repository contains `soc2009/xxx` and `soc2010/xxx` feature branches, used for Google Summer of Code projects.

Support and bugfix branches

In addition to fixing bugs in current master, the Django project provides official bugfix support for the most recent released version of Django, and security support for the two most recently-released versions of Django.

This support is provided via branches in which the necessary bug or security fixes are applied; the branches are then used as the basis for issuing bugfix or security releases.

These branches can be found in the repository as `stable/A.B.x` branches, and new branches will be created there after each new Django release.

For example, shortly after the release of Django 1.0, the branch `stable/1.0.x` was created to receive bug fixes, and shortly after the release of Django 1.1 the branch `stable/1.1.x` was created.

Official support for the above mentioned releases has expired, and so they no longer receive direct maintenance from the Django project. However, the branches continue to exist and interested community members have occasionally used them to provide unofficial support for old Django releases.

10.5.4 Tags

Each Django release is tagged and signed by Django's release manager.

The tags can be found on GitHub's tags page.

CHAPTER

ELEVEN

INDICES, GLOSSARY AND TABLES

- *genindex*
- *modindex*
- *Glossary*

CHAPTER

TWELVE

DEPRECATED/OBSOLETE DOCUMENTATION

The following documentation covers features that have been deprecated or that have been replaced in newer versions of Django.

12.1 Deprecated/obsolete documentation

These documents cover features that have been deprecated or that have been replaced in newer versions of Django. They're preserved here for folks using old versions of Django or those still using deprecated APIs. No new code based on these APIs should be written.

12.1.1 Customizing the Django admin interface

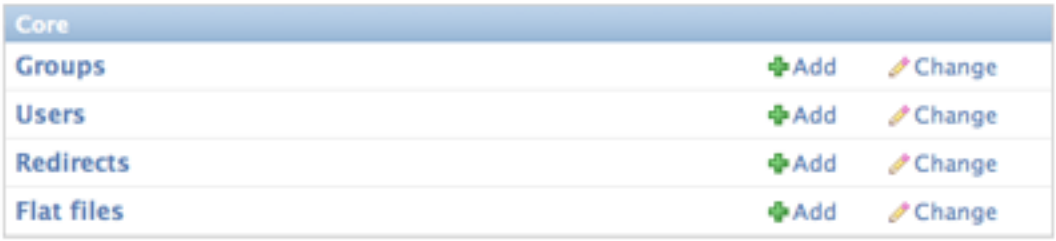
> **Warning:** The design of the admin has changed somewhat since this document was written, and parts may not apply any more. This document is no longer maintained since an official API for customizing the Django admin interface is in development.

Django's dynamic admin interface gives you a fully-functional admin for free with no hand-coding required. The dynamic admin is designed to be production-ready, not just a starting point, so you can use it as-is on a real site. While the underlying format of the admin pages is built in to Django, you can customize the look and feel by editing the admin stylesheet and images.

Here's a quick and dirty overview some of the main styles and classes used in the Django admin CSS.

Modules

The `.module` class is a basic building block for grouping content in the admin. It's generally applied to a `div` or a `fieldset`. It wraps the content group in a box and applies certain styles to the elements within. An `h2` within a `div.module` will align to the top of the `div` as a header for the whole group.



Column Types

Note: All admin pages (except the dashboard) are fluid-width. All fixed-width classes from previous Django versions have been removed.

The base template for each admin page has a block that defines the column structure for the page. This sets a class on the page content area (`div#content`) so everything on the page knows how wide it should be. There are three column types available.

colM This is the default column setting for all pages. The "M" stands for "main". Assumes that all content on the page is in one main column (`div#content-main`).

colMS This is for pages with one main column and a sidebar on the right. The "S" stands for "sidebar". Assumes that main content is in `div#content-main` and sidebar content is in `div#content-related`. This is used on the main admin page.

colSM Same as above, with the sidebar on the left. The source order of the columns doesn't matter.

For instance, you could stick this in a template to make a two-column page with the sidebar on the right:

```
{% block coltype %}colMS{% endblock %}
```

Text Styles

Font Sizes

Most HTML elements (headers, lists, etc.) have base font sizes in the stylesheet based on context. There are three classes are available for forcing text to a certain size in any context.

small 11px

tiny 10px

mini 9px (use sparingly)

Font Styles and Alignment

There are also a few styles for styling text.

.quiet Sets font color to light gray. Good for side notes in instructions. Combine with `.small` or `.tiny` for sheer excitement.

.help This is a custom class for blocks of inline help text explaining the function of form elements. It makes text smaller and gray, and when applied to `p` elements within `.form-row` elements (see Form Styles below), it will offset the text to align with the form field. Use this for help text, instead of `small quiet`. It works on other elements, but try to put the class on a `p` whenever you can.

.align-left It aligns the text left. Only works on block elements containing inline elements.

.align-right Are you paying attention?

.nowrap Keeps text and inline objects from wrapping. Comes in handy for table headers you want to stay on one line.

Floats and Clears

float-left floats left

float-right floats right

clear clears all

Object Tools

Certain actions which apply directly to an object are used in form and changelist pages. These appear in a "toolbar" row above the form or changelist, to the right of the page. The tools are wrapped in a `ul` with the class `object-tools`. There are two custom tool types which can be defined with an additional class on the `a` for that tool. These are `.addlink` and `.viewsitelink`.

Example from a changelist page:

```
<ul class="object-tools">
  <li><a href="/stories/add/" class="addlink">Add redirect</a></li>
</ul>
```

and from a form page:

```
<ul class="object-tools">
 <li><a href="/history/303/152383/">History</a></li>
 <li><a href="/r/303/152383/" class="viewsitelink">View on site</a></li>
</ul>
```

Form Styles

Fieldsets

Admin forms are broken up into groups by `fieldset` elements. Each form fieldset should have a class `.module`. Each fieldset should have a header `h2` within the fieldset at the top (except the first group in the form, and in some cases where the group of fields doesn't have a logical label).

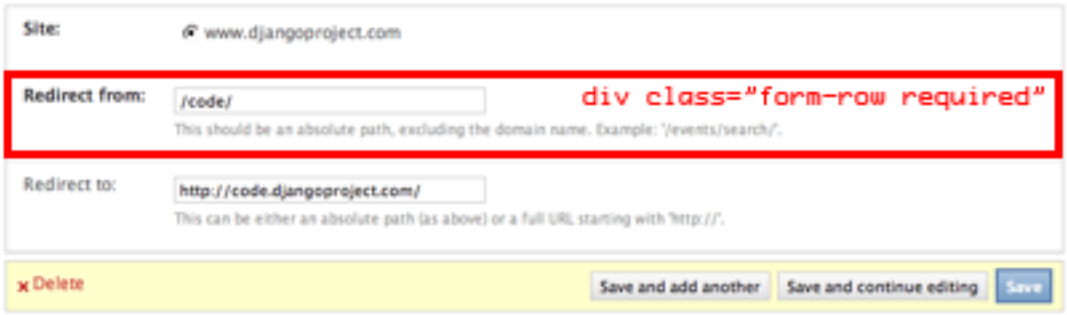
Each fieldset can also take extra classes in addition to `.module` to apply appropriate formatting to the group of fields.

.aligned This will align the labels and inputs side by side on the same line.

.wide Used in combination with `.aligned` to widen the space available for the labels.

Form Rows

Each row of the form (within the `fieldset`) should be enclosed in a `div` with class `form-row`. If the field in the row is required, a class of `required` should also be added to the `div.form-row`.



Labels

Form labels should always precede the field, except in the case of checkboxes and radio buttons, where the `input` should come first. Any explanation or help text should follow the `label` in a `p` with class `.help`.

PYTHON MODULE INDEX

S

t

U

V