# Django Documentation

## version 1.1

Django Software Foundation

August 24, 2009

# Contents

# Django documentation contents

## 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.

### *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*.

#### *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:

```
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
```

#### *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.

#### *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:

```
>>> from mysite.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.

# 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()
```

## 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.

## 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.defaults import *

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', 'mysite.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'mysite.views.month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'mysite.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 `mysite.views.article_detail(request, '2005', '05', '39323')`.

## 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.

## 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/article_detail.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.gif" 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.

## *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!

# *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.

## *Install Python*

Being a Python Web framework, Django requires Python. It works with any Python version from 2.3 to 2.6 (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), but we recommend installing Python 2.5 or later. If you do so, you won't need to set up a database just yet: Python 2.5 or later includes a lightweight database called SQLite.

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's installed by typing `python` from your shell; you should see something like:

```
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

## *Set up a database*

If you installed Python 2.5 or later, you can skip this step for now.

If not, or if you'd like to work with a "large" database engine like PostgreSQL, MySQL, or Oracle, consult the *database installation information*.

## *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*.

## *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.

### Warning

If 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; if you try to use them with an official release they won't work.

## That's it!

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

# 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 by running the Python interactive interpreter and typing `import django`. If that command runs successfully, with no errors, Django is installed.

### 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.

## 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 command `django-admin.py startproject mysite`. This will create a `mysite` directory in your current directory.

### 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 `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/
    __init__.py
    manage.py
    settings.py
    urls.py
```

These files are:

- `__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.)
- `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*.
- `settings.py`: Settings/configuration for this Django project. *Django settings* will tell you all about how settings work.
- `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*.

## The development server

Let's verify this worked. Change into the `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 1.0, 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 `settings.py`. It's a normal Python module with module-level variables representing Django settings. Change these settings to match your database's connection parameters:

- `DATABASE_ENGINE` -- Either 'postgresql_psycopg2', 'mysql' or 'sqlite3'. Other backends are `also available`.
- `DATABASE_NAME` -- The name of your database. If you're using SQLite, the database will be a file on your computer; in that case, `DATABASE_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`).
- `DATABASE_USER` -- Your database username (not used for SQLite).
- `DATABASE_PASSWORD` -- Your database password (not used for SQLite).
- `DATABASE_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 `DATABASE_ENGINE` to `'sqlite3'`). SQLite is included as part of Python 2.5 and later, so you won't need to install anything else.

### 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`, take note of the `INSTALLED_APPS` setting towards the bottom of the file. That variable 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.

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`.

## *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.

In this tutorial, we'll create our poll app in the `mysite` directory, for simplicity. As a consequence, the app will be coupled to the project -- that is, Python code within the poll app will refer to `mysite.polls`. Later in this tutorial, we'll discuss decoupling your apps for distribution.

To create your app, make sure you're in the `mysite` directory 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
    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: polls and choices. 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 = models.CharField(max_length=200)
    votes = models.IntegerField()
```

### *Errors about `max_length`*

If Django gives you an error message saying that `max_length` is not a valid argument, you're most likely using an old version of Django. (This version of the tutorial is written for the latest development version of Django.) If you're using a Subversion checkout of Django's development version (see *the installation docs* for more information), you shouldn't have any problems.

If you want to stick with an older version of Django, you'll want to switch to the Django 0.96 tutorial, because this tutorial covers several features that only exist in the Django development version.

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.

## *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 `'mysite.polls'`. So it'll look like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'mysite.polls'
)
```

Now Django knows `mysite` includes 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"),
    "choice" 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 have 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.

## 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 up the project's environment for you. "Setting up the environment" involves two things:

- Putting `mysite` on `sys.path`. For flexibility, several pieces of Django refer to projects in Python dotted-path notation (e.g. `'mysite.polls.models'`). In order for this to work, the `mysite` package has to be on `sys.path`.

  We've already seen one example of this: the `INSTALLED_APPS` setting is a list of packages in dotted-path notation.
- Setting the `DJANGO_SETTINGS_MODULE` environment variable, which gives Django the path to your `settings.py` file.

### Bypassing manage.py

If you'd rather not use manage.py, no problem. Just make sure mysite is at the root level on the Python path (i.e., import mysite works) and set the DJANGO_SETTINGS_MODULE environment variable to mysite.settings.

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

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

```
>>> from mysite.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.
>>> import datetime
>>> p = Poll(question="What's up?", pub_date=datetime.datetime.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 up?"
>>> p.pub_date
datetime.datetime(2007, 7, 15, 12, 00, 53)

# Change values by changing the attributes, then calling save().
>>> p.pub_date = datetime.datetime(2007, 4, 1, 0, 0)
>>> 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
```

## *If __unicode__() doesn't seem to work*

If you add the __unicode__() method to your models and don't see any change in how they're represented, you're most likely using an old version of Django. (This version of the tutorial is written for the latest development version of Django.) If you're using a Subversion checkout of Django's development version (see *the installation docs* for more information), you shouldn't have any problems.

If you want to stick with an older version of Django, you'll want to switch to the Django 0.96 tutorial, because this tutorial covers several features that only exist in the Django development version.

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 jibberish 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
# ...
class Poll(models.Model):
    # ...
    def was_published_today(self):
        return self.pub_date.date() == datetime.date.today()
```

Note the addition of `import datetime` to reference Python's standard `datetime` module.

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

```
>>> from mysite.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 2007. Of course, if you're going through this
# tutorial in another year, change as appropriate.
>>> Poll.objects.get(pub_date__year=2007)
<Poll: What's up?>

>>> Poll.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Poll matching query does not exist.

# 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_today()
False

# 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.
>>> p = Poll.objects.get(pk=1)
>>> p.choice_set.create(choice='Not much', votes=0)
<Choice: Not much>
>>> p.choice_set.create(choice='The sky', votes=0)
<Choice: The sky>
>>> c = p.choice_set.create(choice='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 2007.
>>> Choice.objects.filter(poll__pub_date__year=2007)
[<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__startswith='Just hacking')
>>> c.delete()
```

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.

# 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 necessarily intended to be used by site visitors; it's for site managers.

## 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:

- Add `"django.contrib.admin"` to your `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 below the "Uncomment the next two lines..." comment. 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:

Changed in version 1.1: The method for adding admin urls has changed in Django 1.1.`from django.conf.urls.defaults import * # Uncomment the next two lines to enable the admin: from django.contrib import admin admin.autodiscover() urlpatterns = patterns('', # Example: # (r'^mysite/', include('mysite.foo.urls')), # Uncomment the admin/doc line below and add 'django.contrib.admindocs' # to INSTALLED_APPS to enable admin documentation: # (r'^admin/doc/', include('django.contrib.admindocs.urls')), # Uncomment the next line to enable the admin: (r'^admin/', include(admin.site.urls)), )` (The bold lines are the ones that needed to be uncommented.)

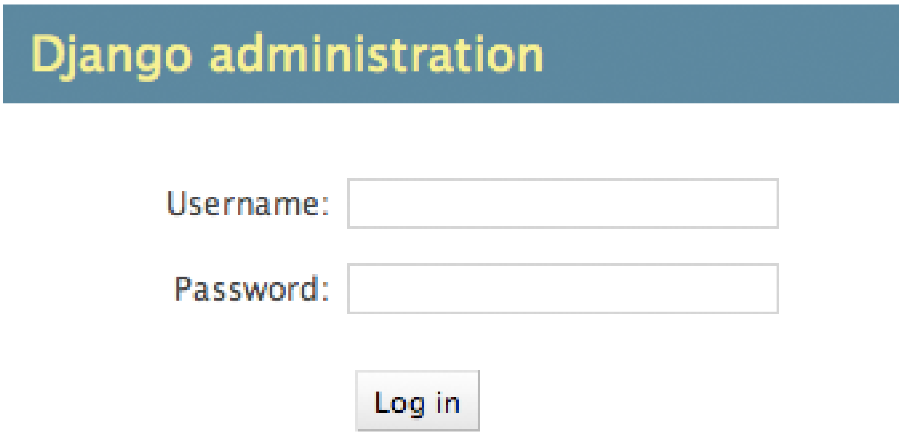
## Start the development server

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

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

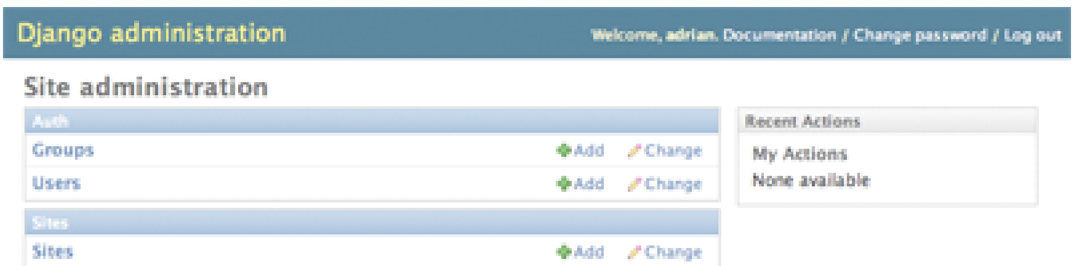
```
python manage.py runserver
```

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:




## 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 other types of editable content, including groups, users and sites. These are core features Django ships with by default.

## 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:

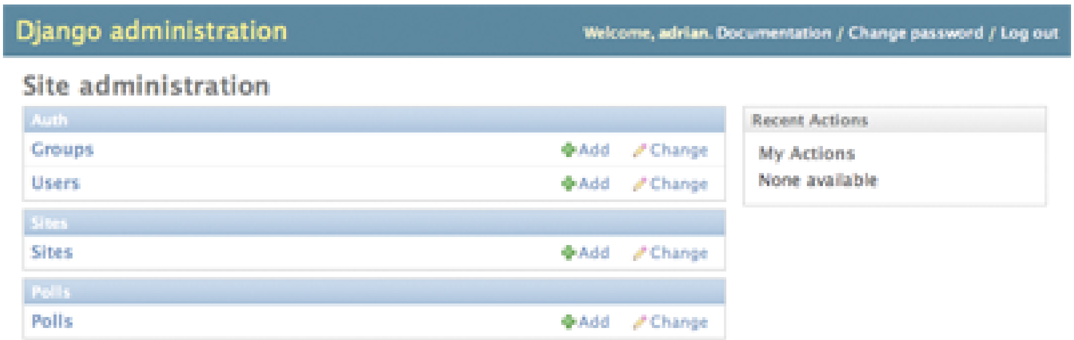
```
from mysite.polls.models import Poll
from django.contrib import admin

admin.site.register(Poll)
```

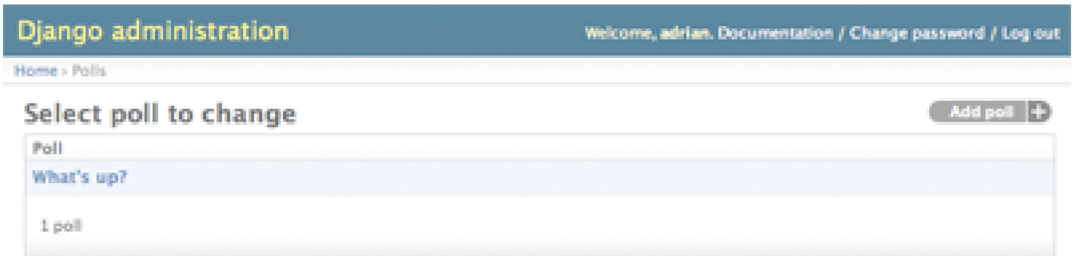
You'll need to restart the development server to see your changes. Normally, the server auto-reloads code every time you modify a file, but the action of creating a new file doesn't trigger the auto-reloading logic.

## Explore the free admin functionality

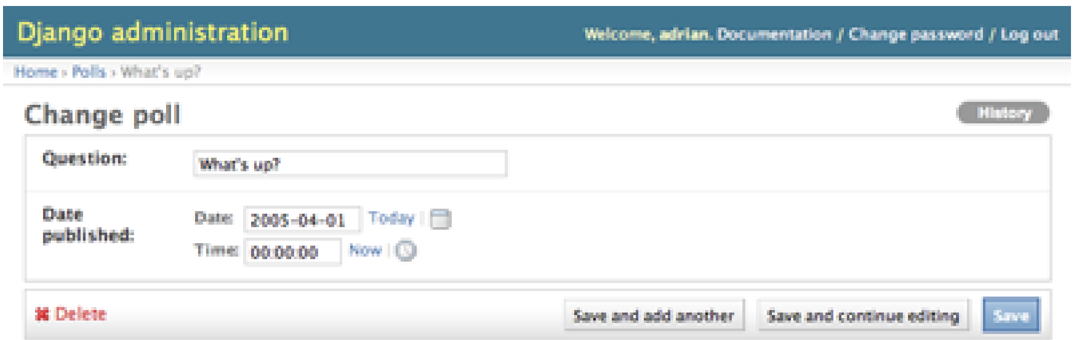
Now that we've registered `Poll`, Django knows that it should be displayed on the admin index page:




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:

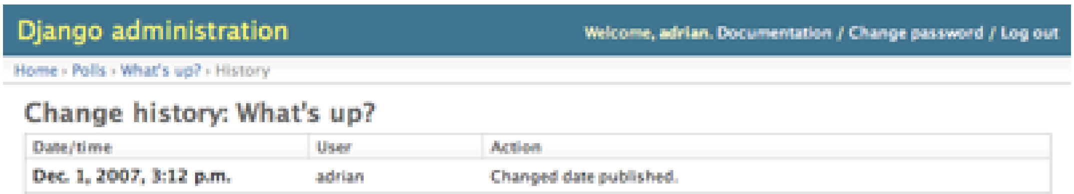
- The form is automatically generated from the Poll model.
- The different model field types (`DateTimeField`, `CharField`) correspond to the appropriate HTML input widget. Each type of field knows how to display itself in the Django admin.
- Each `DateTimeField` gets free JavaScript shortcuts. Dates 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.

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:




## *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.

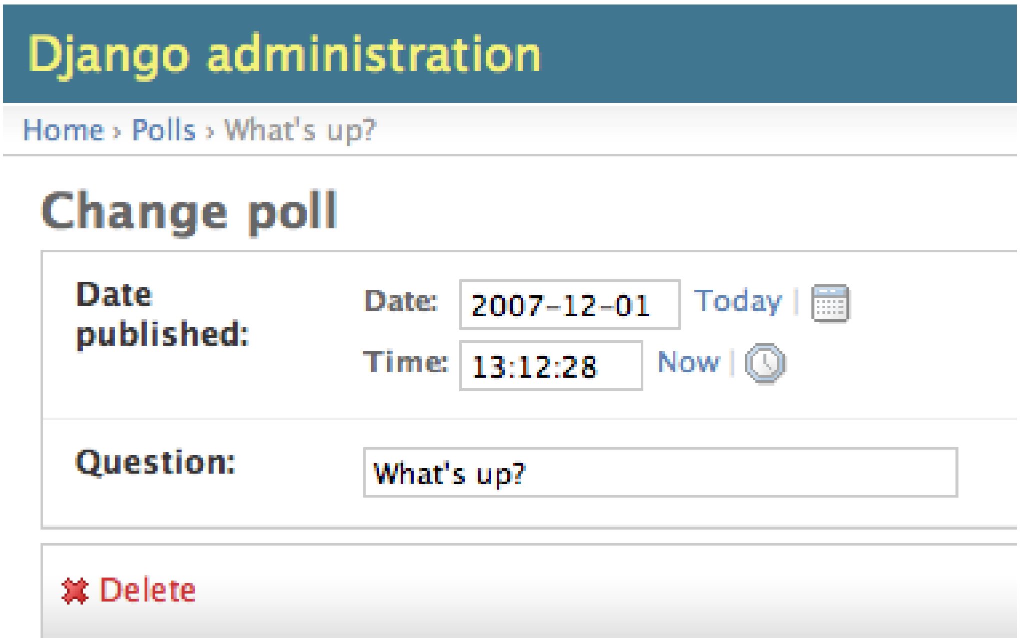
Let's see how this works by re-ordering the fields on the edit form. Replace the `admin.site.register(Poll)` line with:

```
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:



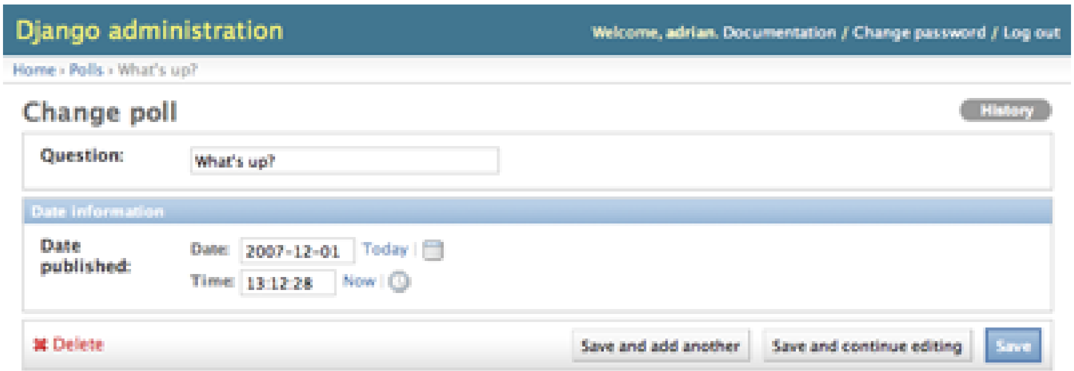

This isn't impressive with only two fields, but for admin forms with dozens of fields, choosing an intuitive order is an important usability detail.

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']}),
    ]
```

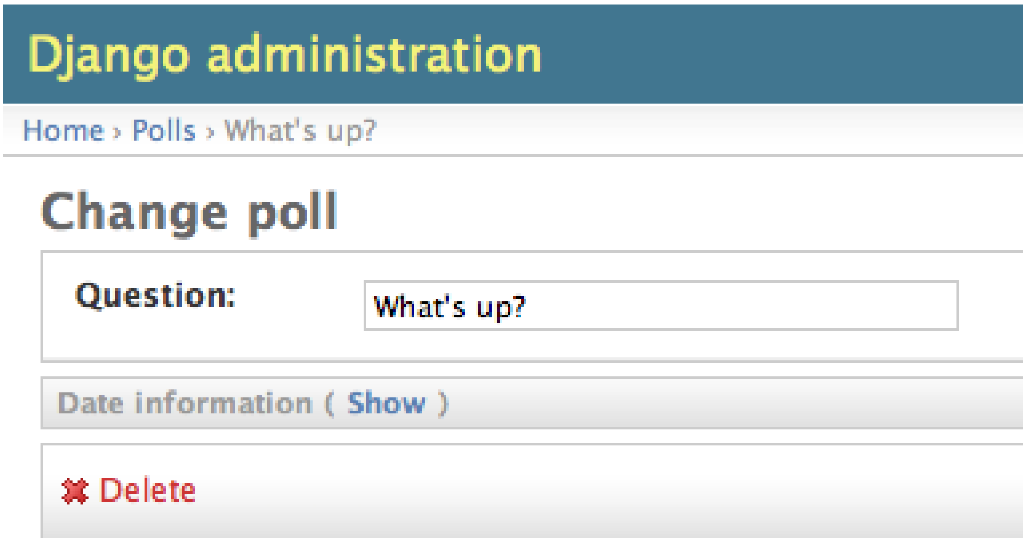
```
admin.site.register(Poll, PollAdmin)
```

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']}),
    ]
```




## Adding related objects

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

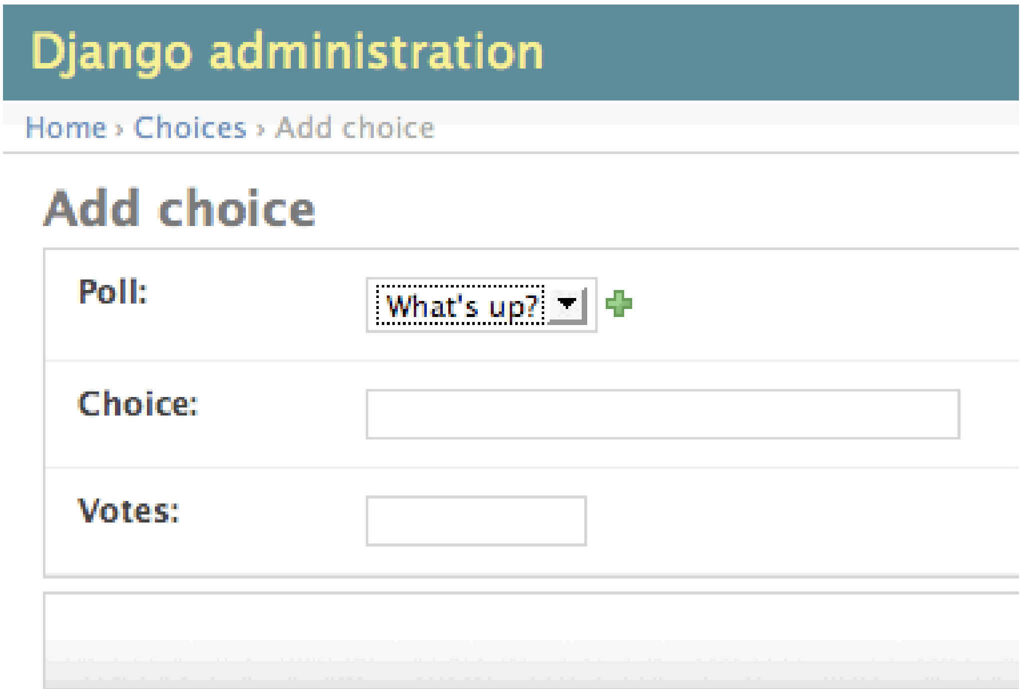
Yet.

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 mysite.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:

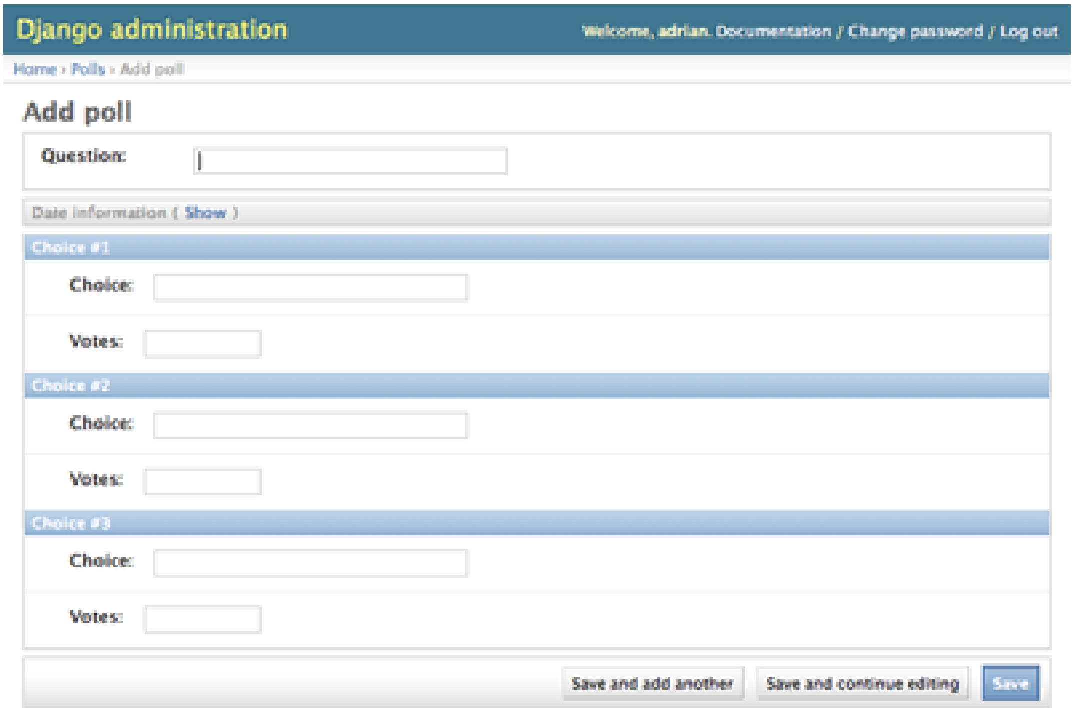
```
class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

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:

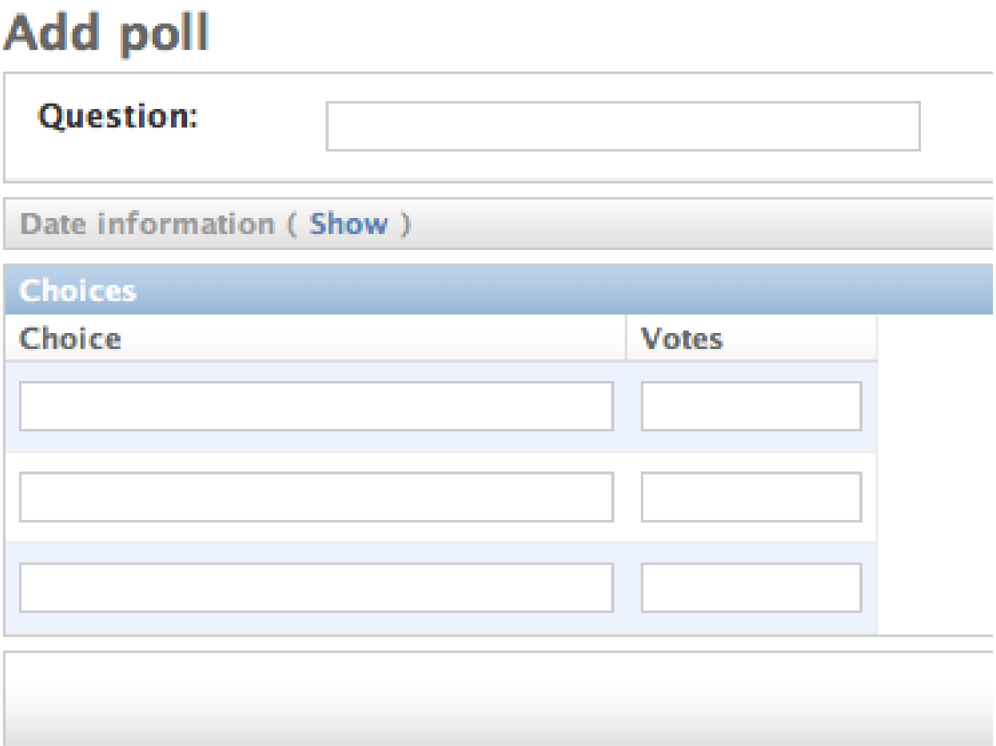



It works like this: There are three slots for related Choices -- as specified by `extra` -- and each time you come back to the "Change" page for an already-created object, you get another three extra slots.

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):
    #...
```

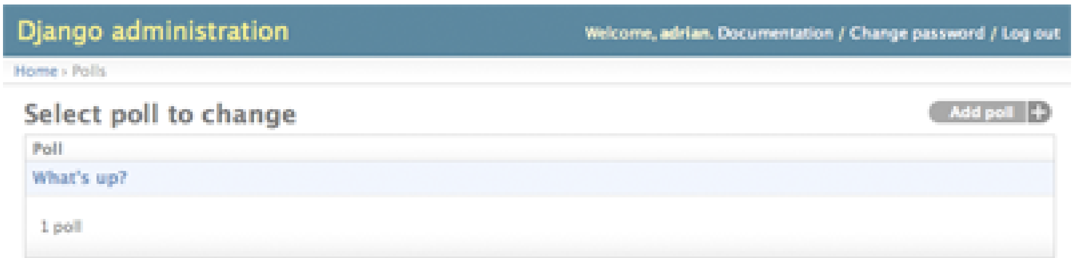
With that `TabularInline` (instead of `StackedInline`), the related objects are displayed in a more compact, table-based format:




## 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:



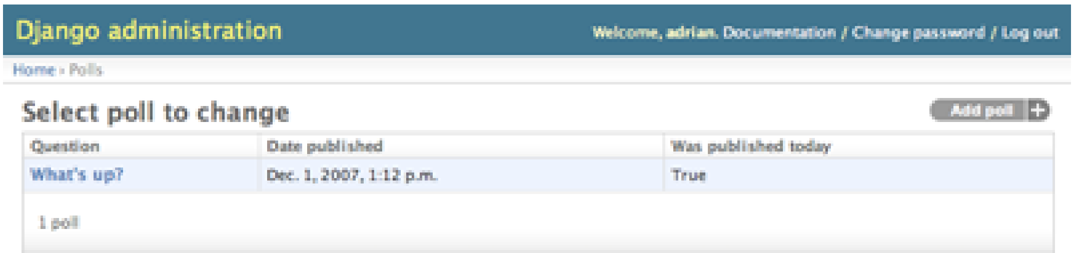

By default, Django displays the `str()` of each object. But sometimes it'd be more helpful if we could display individual fields. To do that, use the `list_display` admin option, which is a tuple of field names to display, as columns, on the change list page for the object:

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

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

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

Now the poll change list page looks like this:



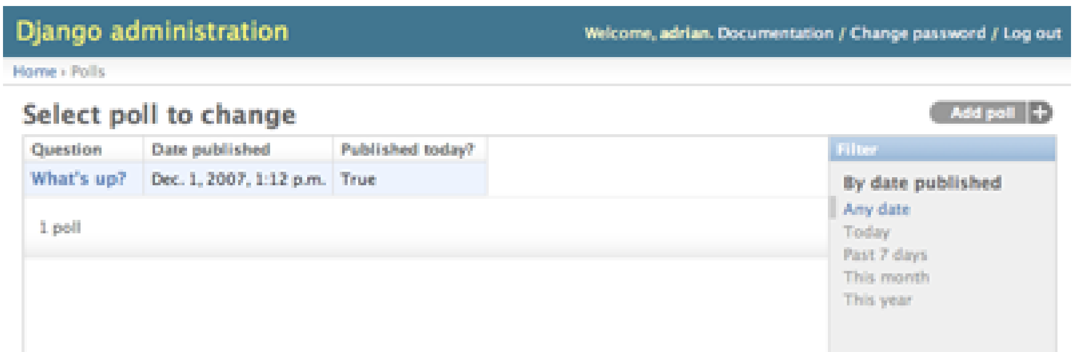

You can click on the column headers to sort by those values -- except in the case of the `was_published_today` header, because sorting by the output of an arbitrary method is not supported. Also note that the column header for `was_published_today` is, by default, the name of the method (with underscores replaced with spaces). But you can change that by giving that method (in `models.py`) a `short_description` attribute:

```
def was_published_today(self):
    return self.pub_date.date() == datetime.date.today()
was_published_today.short_description = 'Published today?'
```

Let's add another 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 the default filter options for DateTimeFields: "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 50 items per page. Change-list pagination, search boxes, filters, date-hierarchies and column-header-ordering all work together like you think they should.

## 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. (How meta!)

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 (`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.

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

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.

## 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.

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

# 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."

## 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 "archive" 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.

## 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.defaults import *

from django.contrib import admin
admin.autodiscover()

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

## 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/

Tried index in module mysite.polls.views. Error was: 'module'
object has no attribute 'index'
```

This error happened because you haven't written a function `index()` in the module `mysite/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 `mysite/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 add the following view. It's slightly different, because it takes 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)
```

Take a look in your browser, at "/polls/34/". It'll display whatever ID you provide in the URL.

## 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 mysite.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 mysite.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>{{ poll.question }}</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.

## *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 mysite.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.

## 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 module's `get()` function. 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.

## 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`, 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. By default, URLconfs have the following line up top:

```
from django.conf.urls.defaults import *
```

That takes care of setting `handler404` in the current module. As you can see in `django/conf/urls/defaults.py`, `handler404` is set to `django.views.defaults.page_not_found()` by default.

Four 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.
- If you don't define your own 404 view -- and simply use the default, which is recommended -- you still have one obligation: To 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`.

## Write a 500 (server error) view

Similarly, URLconfs 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.

## 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 }}</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 attribute lookup -- which works, in this case. If attribute lookup had failed, it would've tried calling the method `question()` on the poll object.

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.

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

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

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

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

```
...
urlpatterns = patterns('',
    (r'^polls/', include('mysite.polls.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 'mysite.polls.urls' URLconf for further processing.

Now that we've decoupled that, we need to decouple the 'mysite.polls.urls' URLconf by removing the leading "polls/" from each line, and removing the lines registering the admin site:

```
urlpatterns = patterns('mysite.polls.views',
    (r'^$', 'index'),
    (r'^(?P<poll_id>\d+)/$', 'detail'),
    (r'^(?P<poll_id>\d+)/results/$', 'results'),
    (r'^(?P<poll_id>\d+)/vote/$', '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 URL root, and the app will still work.

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

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

# 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.

## 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">
{% 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 }}</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

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'),
```

So let's create a `vote()` function in `mysite/polls/views.py`:

```
from django.shortcuts import get_object_or_404, render_to_response
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from mysite.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.",
        })
    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('mysite.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 }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>
```

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.

## 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. Rename a few templates.
3. Delete some the old, now unneeded views.
4. 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.defaults import *

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

Change it like so:

```
from django.conf.urls.defaults import *
from mysite.polls.models import Poll

info_dict = {
    'queryset': Poll.objects.all(),
}

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.list_detail.object_list', info_dict),
    (r'^(?P<object_id>\d+)/$', 'django.views.generic.list_detail.object_detail', info_dict),
    url(r'^(?P<object_id>\d+)/results/$', 'django.views.generic.list_detail.object_detail', dict(info_dict, template_name='polls/results.html'), 'poll_results'),
    (r'^(?P<poll_id>\d+)/vote/$', 'mysite.polls.views.vote'),
)
```

We're using two generic views here: object_list() and object_detail(). 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 data it will be acting upon. This data is provided in a dictionary. The queryset key in this dictionary points to the list of objects to be manipulated by the generic view.
- The object_detail() generic view expects the ID value captured from the URL to be called "object_id", so we've changed poll_id to object_id 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.defaults here. It's a good habit to use url() when you are providing a pattern name like this.

By default, the object_detail() generic view uses a template called <app name>/<model name>_detail.html. In our case, it'll use the template "polls/poll_detail.html". Thus, rename your polls/detail.html template to polls/poll_detail.html, and change the render_to_response() line in vote().

Similarly, the object_list() generic view uses a template called <app name>/<model name>_list.html. Thus, rename polls/index.html to polls/poll_list.html.

Because we have more than one entry in the URLconf that uses object_detail() for the polls app, we manually specify a template name for the results view: template_name='polls/results.html'. Otherwise, both views would use the same template. Note that we use dict() to return an altered dictionary in place.

> ## Note
>
> django.db.models.QuerySet.all() is lazy
>
> It might look a little frightening to see Poll.objects.all() being used in a detail view which only needs one Poll object, but don't worry; Poll.objects.all() is actually a special object called a QuerySet, which is "lazy" and doesn't hit your database until it absolutely has to. By the time the database query happens, the object_detail() generic view will have narrowed its scope down to a single object, so the eventual query will only select one row from the database.
>
> If you'd like to know more about how that works, The Django database API documentation *explains the lazy nature of QuerySet objects*.

In previous parts of the tutorial, the templates have been provided with a context that contains the poll and latest_poll_list context variables. However, the generic views provide the variables object and object_list as context.

Therefore, you need to change your templates to match the new context variables. Go through your templates, and modify any reference to `latest_poll_list` to `object_list`, and change any reference to `poll` to `object`.

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 `vote()` view is still required. However, it must be modified to match the new context variables. In the `render_to_response()` call, rename the `poll` context variable to `object`.

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*.

## 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*

# 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.)

## 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 Page* and the *Index*.

Or you can just browse around!

## 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 spent 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*.

## 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 django/trunk/docs directory of our Subversion repository. Each document online is a separate text file in the repository.

## 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 http://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 the Subversion "trunk"), note that the `docs/` directory contains all of the documentation. You can `svn update` it, just as you `svn update` the Python code, in order to get the latest changes.

You can check out the latest Django documentation from Subversion using this shell command:

```
$ svn co http://code.djangoproject.com/svn/django/trunk/docs/ django_docs
```

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 website, or by Python's `easy_install`:

  ```
  $ easy_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.
* The HTML documentation will be placed in `docs/_build/html`.

> **Warning**
>
> At the time of this writing, Django's using a version of Sphinx not yet released, so you'll currently need to install Sphinx from the source. We'll fix this shortly.

## Differences between versions

As previously mentioned, the text documentation in our Subversion repository contains the "latest and greatest" changes and additions. These changes often include documentation of new features added in the Django development version -- the Subversion ("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 Subversion. 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 Subversion 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.

# Using Django

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

## How to install Django

This document will get you up and running with Django.

## Install Python

Being a Python Web framework, Django requires Python.

It works with any Python version from 2.3 to 2.6 (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.

## 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 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. 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. A great second choice is *mod_python*, the predecessor to mod_wsgi. Additionally, Django follows the WSGI spec, which allows it to run on a variety of server platforms. See the server-arrangements wiki page for specific installation instructions for each platform.

## 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 psycopg package. Django supports both version 1 and 2. (When you configure Django's database layer, specify either `postgresql` [for version 1] or `postgresql_psycopg2` [for version 2].)

  If you're on Windows, check out the unofficial compiled Windows version.
- If you're using MySQL, you'll need MySQLdb, version 1.2.1p2 or higher. You will also want to read the database-specific notes for the *MySQL backend*.
- If you're using SQLite and either Python 2.3 or Python 2.4, you'll need pysqlite. Use version 2.0.3 or higher. Python 2.5 ships with an SQLite wrapper in the standard library, so you don't need to install anything extra in that case. Please read the SQLite backend *notes*.
- 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.

## 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 `setup.py install`, uninstalling is as simple as deleting the `django` directory from your Python `site-packages`.

If you installed Django from a Python egg, remove the Django `.egg` file, and remove the reference to the egg in the file named `easy-install.pth`. This file should also be located in your `site-packages` directory.

### Where are my `site-packages` stored?

The location of the `site-packages` directory depends on the operating system, and the location in which Python was installed. To find out 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.)

## 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

1. Download the latest release from our download page.
2. Untar the downloaded file (e.g. `tar xzvf Django-NNN.tar.gz`, where `NNN` 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-NNN`).
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 up a command shell with administrator privileges and run the command `setup.py install`.

These commands will install Django in your Python installation's `site-packages` directory.

### 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 Subversion installed, and that you can run its commands from a shell. (Enter `svn help` at a shell prompt to test this.)
2. Check out Django's main development branch (the 'trunk') like so:

```
svn co http://code.djangoproject.com/svn/django/trunk/ django-trunk
```

3. Next, make sure that the Python interpreter can load Django's code. There are various ways of accomplishing this. One of the most convenient, on Linux, Mac OSX or other Unix-like systems, is to use a symbolic link:

```
ln -s `pwd`/django-trunk/django SITE-PACKAGES-DIR/django
```

   (In the above line, change `SITE-PACKAGES-DIR` to match the location of your system's `site-packages` directory, as explained in the "Where are my `site-packages` stored?" section above.)

   Alternatively, you can define your `PYTHONPATH` environment variable so that it includes the `django-trunk` directory. This is perhaps the most convenient solution on Windows systems, which don't support symbolic links. (Environment variables can be defined on Windows systems from the Control Panel.)

   ### What about Apache and mod_python?

   If you take the approach of setting `PYTHONPATH`, you'll need to remember to do the same thing in your Apache configuration once you deploy your production site. Do this by setting `PythonPath` in your Apache configuration file.

   More information about deployment is available, of course, in our *How to use Django with mod_python* documentation.

4. 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 `pwd`/django-trunk/django/bin/django-admin.py /usr/local/bin
```

   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:\Python24\Scripts`.

You *don't* have to run `python setup.py install`, because you've already carried out the equivalent actions in steps 3 and 4.

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

# 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.

## 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*.

## See also

A companion to this document is the official repository of model examples. (In the Django source distribution, these examples are in the `tests/modeltests` directory.)

## 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 = (
    (u'FR', u'Freshman'),
    (u'SO', u'Sophomore'),
    (u'JR', u'Junior'),
    (u'SR', u'Senior'),
    (u'GR', u'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):
    GENDER_CHOICES = (
        (u'M', u'Male'),
        (u'F', u'Female'),
    )
    name = models.CharField(max_length=60)
    gender = models.CharField(max_length=2, choices=GENDER_CHOICES)
```

```
>>> p = Person(name="Fred Flinstone", gender="M")
>>> p.save()
>>> p.gender
u'M'
>>> p.get_gender_display()
u'Male'
```

**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 `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*

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

`ForeignKey` 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.

## *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-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 `ManyToManyField` (`toppings` in the example above) be a plural describing the set of related model objects.

It doesn't matter which model gets the `ManyToManyField`, but you only need it in one of the models -- not in 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

New in version 1.0: *Please, see the release notes*

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]
```

## *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.

New in version 1.0: *Please, see the release notes*

`OneToOneField` fields also accept one optional 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 model that holds your model. Then, just refer to the other model class wherever needed. For example:

```
from mysite.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

New in version 1.0: *Please, see the release notes*

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*. Read more about properties.

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

It's important to remember to call the superclass method -- that's that `super(Blog, self).save()` 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.

## 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

New in version 1.0: *Please, see the release notes*

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 behaviour 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 be the string `'%(class)s'`. This is replaced by the lower-cased name of the child class that the field is used in. Since each class has a different name, each related name will end up being different. For example:

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

    class Meta:
        abstract = True

class ChildA(Base):
```

```
    pass

class ChildB(Base):
    pass
```

The reverse name of the `ChildA.m2m` field will be `childa_related`, whilst the reverse name of the `ChildB.m2m` field will be `childb_related`. It is up to you how you use the `'%(class)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.filter(name="Bob's Cafe")
# If Bob's Cafe 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 `django.db.models.Options.ordering` attribute or a `django.db.models.Options.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 django.db.models.fields.ForeignKey and django.db.models.fields.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

New in version 1.1: *Please, see the release notes*

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 unorderd 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 initialising new instances (specifying which field is being intialised 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 merely 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` exception if you override any model field in any ancestor model.

## *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()
    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 mysite.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 an object and save it all in one step see 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 `ForeignKey` fields works exactly the same way as saving a normal field; simply assign an object of the right type to the field in question:

```
>>> 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:

```
>> joe = Author.objects.create(name="Joe")
>> entry.authors.add(joe)
```

Django will complain if you try to assign or add an object of the wrong type.

## *Retrieving objects*

To retrieve objects from your database, you 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*.

### Other QuerySet methods

Most of the time you'll use all(), 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.

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__author__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__author__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__author__isnull=False,
        entry__author__name__isnull=True)
```

## *Spanning multi-valued relationships*

New in version 1.0: *Please, see the release notes*

When you are filtering an object based on a `ManyToManyField` or a reverse `ForeignKeyField`, 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

New in version 1.1: *Please, see the release notes*

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() object 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 comment count, and use that F() object in the query:

```
>>> from django.db.models import F
>>> Entry.objects.filter(n_pingbacks__lt=F('n_comments'))
```

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 *twice* as many comments as pingbacks, we modify the query:

```
>>> Entry.objects.filter(n_pingbacks__lt=F('n_comments') * 2)
```

To find all the entries where the sum of the pingback count and comment count is greater than the rating of the entry, 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(author__name=F('blog__name'))
```

## 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 = Poll.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:

```
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, 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()
```

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()
```

## Updating multiple objects at once

New in version 1.0: *Please, see the release notes*

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. 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 doesn't return anything (similar to `delete()`). The only restriction on the `QuerySet` that is updated is that it can only access one database table, the model's main table. So don't try to filter based on related fields or anything like that; it won't work.

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()
```

New in version 1.1: *Please, see the release notes*

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 into raw-SQL statement mode.

The preferred way to do this is by giving your model custom methods or custom manager methods that execute queries. Although there's nothing in Django that *requires* database queries to live in the model layer, this approach keeps all your data-access logic in one place, which is smart from an code-organization standpoint. For instructions, 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.

# Aggregation

New in version 1.1: *Please, see the release notes*

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)
```

## 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 preceeds 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}
```

## *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 this first `Manager` defined in a class as the "default" `Manager`, and several parts of Django (though not the admin application) will use that `Manager` exclusively for that model. As a result, it's often a good idea to be careful in your choice of default manager, in order to avoid a situation where overriding of `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 = CustomerManager()

    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.
```

## 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 be remember and that's the topic of this section.

### 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*, object 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.

## Performing raw SQL queries

Feel free to write custom SQL statements in custom model methods and module-level methods. The object `django.db.connection` represents the current database connection, and `django.db.transaction` represents the current 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(self):
    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
```

## Transactions and raw SQL

If you are using transaction decorators (such as `commit_on_success`) to wrap your views and provide transaction control, you don't have to make a manual call to `transaction.commit_unless_managed()` -- you can manually commit if you want to, but you aren't required to, since the decorator will commit for you. However, if you don't manually commit your changes, you will need to manually mark the transaction as dirty, using `transaction.set_dirty()`:

```
@commit_on_success
def my_custom_sql_view(request, value):
    from django.db import connection, transaction
    cursor = connection.cursor()

    # Data modifying operation
    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [value])

    # Since we modified data, mark the transaction as dirty
    transaction.set_dirty()

    # Data retrieval operation. This doesn't dirty the transaction,
    # so no call to set_dirty() is required.
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [value])
    row = cursor.fetchone()

    return render_to_response('template.html', {'row': row})
```

The call to `set_dirty()` is made automatically when you use the Django ORM to make data modifying database calls. However, when you use raw SQL, Django has no way of knowing if your SQL modifies data or not. The manual call to `set_dirty()` ensures that Django knows that there are modifications that must be committed.

## Connections and cursors

`connection` and `cursor` mostly implement the standard Python DB-API (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.)

## An easier option?

A final note: If all you want to do is a custom `WHERE` clause, you can just use the `where`, `tables` and `params` arguments to the *extra clause* in the standard queryset API.

# 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.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.CacheMiddleware',
    'django.middleware.transaction.TransactionMiddleware',
)
```

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.

An exception is `CacheMiddleware`, which is never affected. The cache middleware uses its own database cursor (which is mapped to its own database connection internally).

## *Controlling transaction management in views*

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 Python decorators to change the way transactions are handled by a particular view function.

> **Note**
>
> Although the examples below use view functions as examples, these decorators can be applied to non-view functions as well.

### *django.db.transaction.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):
    ....
```

Within `viewfunc()`, transactions will be committed as soon as you call `model.save()`, `model.delete()`, or any other function that writes to the database.

### *django.db.transaction.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):
    ....
```

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.

### *django.db.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()
```

### An important note to users of earlier Django releases:

The database `connection.commit()` and `connection.rollback()` methods (called `db.commit()` and `db.rollback()` in 0.91 and earlier) no longer exist. They've been replaced by `transaction.commit()` and `transaction.rollback()`.

## 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 to the PostgreSQL 8 and Oracle backends. Other backends will provide the savepoint functions, but they are empty operations - they won't actually do anything.

Savepoints aren't especially useful if you are using the default `autocommit` behaviour 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()`.

Savepoints are controlled by three methods on the transaction object:

**`transaction.savepoint ()`**
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`*`)`**
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`*`)`**

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

New in version 1.1: *Please, see the release notes*

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.

# Handling HTTP requests

Information on handling HTTP requests in Django:

## 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.

### 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`, 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.defaults.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.

## Example

Here's a sample URLconf:

```
from django.conf.urls.defaults import *

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:

- `from django.conf.urls.defaults import *` makes the `patterns()` function available.
- 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/2003/03/3/` would match the final pattern. Django would call the function `news.views.article_detail(request, '2003', '03', '3')`.

## 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})/$', 'news.views.month_archive'),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d+)/$', '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/2003/03/3/` would call the function `news.views.article_detail(request, year='2003', month='03', day='3')`.

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.defaults.patterns()`. Always use `patterns()` to create the `urlpatterns` variable.

Convention is to use `from django.conf.urls.defaults import *` at the top of your URLconf. This gives your module access to these objects:

### 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*

New in version 1.0: *Please, see the release notes*

**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.

## *handler404*

**handler404**

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 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.

## *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.

New in version 1.1: *Please, see the release notes*

`include()` also accepts as an argument an iterable that returns URL patterns.

See Including other URLconfs below.

## *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.defaults import *

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

In this example, each view has a common prefix -- `'mysite.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.defaults import *

urlpatterns = patterns('mysite.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.defaults import *

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index'),
    (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$', 'django.views.generic.date_based.archive_month'),
    (r'^tag/(?P<tag>\w+)/$', 'weblog.views.tag'),
)
```

New:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('django.views.generic.date_based',
    (r'^$', 'archive_index'),
    (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$','archive_month'),
)

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 the URLconf for the Django Web site itself. It includes a number of other URLconfs:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^weblog/',        include('django_website.apps.blog.urls.blog')),
    (r'^documentation/', include('django_website.apps.docs.urls.docs')),
    (r'^comments/',      include('django.contrib.comments.urls')),
)
```

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.

New in version 1.1: *Please, see the release notes*

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.defaults import *

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 *generic views* and 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*

New in version 1.0: *Please, see the release notes*

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* URLpatterns 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*

New in version 1.1: *Please, see the release notes*

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 first 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 first registered instance of `myapp`. Since there is no default instance, the first 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`.

## *Utility methods*

### *reverse()*

If you need to use something similar to the `url` template tag in your code, Django provides the following method (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.

New in version 1.1: *Please, see the release notes*

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*.

## 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.

### 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() above, you don't need to worry about the urlconf parameter. The function returns the triple (view function, arguments, keyword arguments).

For example, it can be used for testing if 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.

## 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*, let's create a file called `views.py` in the `mysite` directory, which you created in the previous chapter.

### 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`, which lives in 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

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.

This page_not_found view should suffice for 99% of Web applications, but if you want to override the 404 view, 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. By default, URLconfs contain the following line:

```
from django.conf.urls.defaults import *
```

That takes care of setting handler404 in the current module. As you can see in django/conf/urls/defaults.py, handler404 is set to 'django.views.defaults.page_not_found' by default.

Three 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 default 404 view will use that template 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 404.
- 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 the traceback will be displayed instead.

### 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 error view by looking for `handler500`. By default, URLconfs contain the following line:

```
from django.conf.urls.defaults import *
```

That takes care of setting `handler500` in the current module. As you can see in `django/conf/urls/defaults.py`, `handler500` is set to `'django.views.defaults.server_error'` by default.

## File Uploads

New in version 1.0: *Please, see the release notes*

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

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:

**`UploadedFile.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.

**`UploadedFile.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.

**`UploadedFile.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.

**`UploadedFile.name`**
The name of the uploaded file (e.g. `my_file.txt`).

**`UploadedFile.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):
    destination = open('some/file/name.txt', 'wb+')
    for chunk in f.chunks():
        destination.write(chunk)
    destination.close()
```

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

*class* **UploadedFile**

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.

## *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, charset)`**

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`., `None` otherwise.

`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, encoding)`**

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.

# 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_to_response

**render_to_response** (*template*[, *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**

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 `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**

New in version 1.0: *Please, see the release notes*

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*)

New in version 1.1: *Please, see the release notes*

Returns an HttpResponseRedirect to the apropriate 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:

   ```
   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** (*object*, **args*, ***kwargs*)
Calls `get()` on a given model manager, but it raises `django.http.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()`, an `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 `django.http.Http404` if the resulting list is empty.

### *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 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
```

## Generic views

See *Generic views*.

## 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.

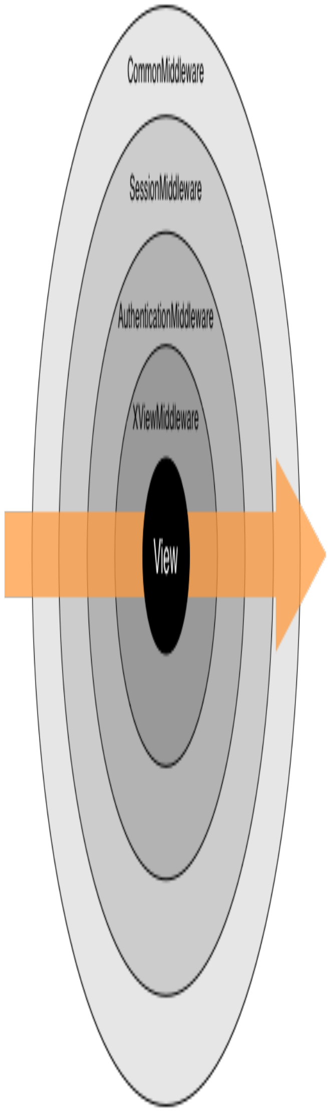
This document explains how middleware works, how you activate middleware, and how to write your own middleware. Django ships with some built-in middleware you can use right out of the box; they're documented in the *built-in middleware reference*.

### 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.contrib.auth.middleware.AuthenticationMiddleware',
)
```

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.

### *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 return 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 starts up.

### 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.

## 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.

## 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 `MIDDLEWARE_CLASSES` contains `'django.contrib.sessions.middleware.SessionMiddleware'`. The default `settings.py` created by `django-admin.py startproject` has `SessionMiddleware` activated.
- Add `'django.contrib.sessions'` to your `INSTALLED_APPS` setting, and run `manage.py syncdb` to install the single database table that stores session data.

Changed in version 1.0: This step is optional if you're not using the database session backend; see configuring the session engine.

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

New in version 1.0: *Please, see the release notes*

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 cached sessions

For better performance, you may want to use a cache-based session backend.

Changed in version 1.1: Django 1.0 did not include the cached_db 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.

### 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 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 it.

A session object 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')`
- `keys()`
- `items()`
- `setdefault()`
- `clear()`

New in version 1.0: `setdefault()` and `clear()` are new in this version.

It also has these methods:

- `flush()`

  New in version 1.0: *Please, see the release notes*

  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)`

  New in version 1.0: *Please, see the release notes*

  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.
- `get_expiry_age()`

  New in version 1.0: *Please, see the release notes*

  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 `settings.SESSION_COOKIE_AGE`.

* `get_expiry_date()`

  New in version 1.0: *Please, see the release notes*

  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 `settings.SESSION_COOKIE_AGE` seconds from now.
* `get_expire_at_browser_close()`

  New in version 1.0: *Please, see the release notes*

  Returns either `True` or `False`, depending on whether the user's session cookie will expire when the user's Web browser is closed.

You can edit `request.session` at any point in your view. You can edit it multiple times.

## 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 `request.session.flush()`. 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 `request.session.set_test_cookie()` in a view, and call `request.session.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

New in version 1.0: *Please, see the release notes*

An API is available to manipulate session data outside of a view:

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceead')
>>> s['last_login'] = datetime.datetime(2005, 8, 20, 13, 35, 10)
>>> s['last_login']
datetime.datetime(2005, 8, 20, 13, 35, 0)
>>> s.save()
```

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`. If `SESSION_SAVE_EVERY_REQUEST` is `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.

## 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.

New in version 1.0: *Please, see the release notes*

This setting is a global default and can be overwritten at a per-session level by explicitly calling `request.session.set_expiry()` 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

New in version 1.0: *Please, see the release notes*

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'`

See configuring the session engine for more details.

### SESSION_FILE_PATH

New in version 1.0: *Please, see the release notes*

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 `".lawrence.com"` (note the leading dot!) for cross-domain cookies, or use `None` for a standard domain cookie.

### *SESSION_COOKIE_NAME*

Default: `'sessionid'`

The name of the cookie to use for sessions. This can be whatever you want.

### *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.

# *Working with forms*

## *About this document*

This document provides an introduction to Django's form handling features. For a more detailed look at the forms API, see *The Forms API*. For documentation of the available field types, see *Form fields*.

`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.

## 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 e-mail 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).

## 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:

```
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_to_response('contact.html', {
        'form': form,
    })
```

There are three code paths here:

1. If the form has not been submitted, an unbound instance of ContactForm is created and passed to the template.
2. If the form has been submitted, a bound instance of the form is created using `request.POST`. If the submitted data is valid, it is processed and the user is re-directed to a "thanks" page.
3. If the form has been submitted but is invalid, the bound form instance is passed on to the template.

Changed in version 1.0: The `cleaned_data` attribute was called `clean_data` in earlier releases.

The distinction between **bound** and **unbound** forms is important. An unbound form does not have any data associated with it; when rendered to the user, it will be empty or will contain default values. A bound form does have 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 where they went wrong.

See *Bound and unbound forms* for further information on the differences between bound and unbound forms.

## Handling file uploads with a form

To see how to handle file uploads with your form see *Binding uploaded files to a form* for more information.

## Processing the data from a form

Once `is_valid()` returns `True`, you can process the form submission safe in the knowledge that it conforms to the validation rules defined by your form. While you could access `request.POST` directly at this point, it is better to access `form.cleaned_data`. This data has not only been validated but will also be converted in to the relevant Python types for you. 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.

Extending the above 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
```

For more on sending e-mail from Django, see *Sending e-mail*.

## 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">
{{ 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.

`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">
    <div class="fieldWrapper">
        {{ form.subject.errors }}
        <label for="id_subject">E-mail 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. `E-mail address`.

**`{{ field.label_tag }}`**
: The field's label wrapped in the appropriate HTML `<label>` tag, e.g. `<label for="id_email">E-mail address</label>`

**`{{ 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">
    {% for field in form.visible_fields %}
        <div class="fieldWrapper">

            {# Include the hidden fields in the form #}
            {% if forloop.first %}
                {% for hidden in form.hidden_fields %}
                {{ hidden }}
                {% endfor %}
            {% endif %}

            {{ 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.

New in version 1.1: The `hidden_fields` and `visible_fields` methods are new in Django 1.1.

## 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` tag:

```
<form action="/comments/add/" method="POST">
    {% with comment_form as form %}
        {% include "form_snippet.html" %}
    {% endwith %}
    <p><input type="submit" value="Submit comment" /></p>
</form>
```

If you find yourself doing this often, you might consider creating a custom *inclusion tag*.

## Further topics

This covers the basics, but forms can do a whole lot more:

## Creating forms from models

### 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 |
| `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` |
| `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` with `verify_exists` set to the model field's `verify_exists` |
| `XMLField` | `CharField` with `widget=forms.Textarea` |

New in version 1.0: The `FloatField` form field and `DecimalField` model and form fields are new in Django 1.0.

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 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.):

```
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 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`.

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.

   Changed in version 1.1: *Please, see the release notes*

   The form will render the fields in the same order they are specified in the `fields` attribute.
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. 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

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 for a given model field. You 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 widget, then specify the `widget` parameter when declaring the form field:

```
>>> class ArticleForm(ModelForm):
...     pub_date = DateField(widget=MyDateWidget())
...
...     class Meta:
...         model = Article
```

## Changing the order of fields

New in version 1.1: *Please, see the release notes*

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.

## 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" name="form-INITIAL_FORMS" value="0" id="id_form-INITIAL_FORMS" />
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name" type="text" name="form-0-name" maxlength="100" /></td></tr>
<tr><th><label for="id_form-0-title">Title:</label></th><td><select name="form-0-title" id="id_form-0-title">
<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-birth_date" id="id_form-0-birth_date" /><input type="hidden" name="form-0-id" id="id_form-0-id" /></td></tr>
```

> **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):
        self.queryset = Author.objects.filter(name__startswith='O')
        super(BaseAuthorFormSet, self).__init__(*args, **kwargs)
```

Then, pass your `BaseAuthorFormSet` class to the factory function:

```
>>> AuthorFormSet = modelformset_factory(Author, formset=BaseAuthorFormSet)
```

## *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',))
```

## *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).

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` parameter to `modelformset_factory` to limit the number of forms displayed. With model formsets, this property limits the query to select only the maximum number of objects needed:

```
>>> Author.objects.order_by('name')
[<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]

>>> AuthorFormSet = modelformset_factory(Author, max_num=2, extra=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> formset.initial
[{'id': 1, 'name': u'Charles Baudelaire'}, {'id': 3, 'name': u'Paul Verlaine'}]
```

If the value of `max_num` is higher than the number of objects returned, 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.forms:
...     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="form-0-name" value="Charles Baudelaire" maxlength="100" /><input type="hidden" name="form-0-id" value="1" id="id_form-0-id" /></td></tr>
<tr><th><label for="id_form-1-name">Name:</label></th><td><input id="id_form-1-name" type="text" name="form-1-name" value="Paul Verlaine" maxlength="100" /><input type="hidden" name="form-1-id" value="3" id="id_form-1-id" /></td></tr>
<tr><th><label for="id_form-2-name">Name:</label></th><td><input id="id_form-2-name" type="text" name="form-2-name" value="Walt Whitman" maxlength="100" /><input type="hidden" name="form-2-id" value="2" id="id_form-2-id" /></td></tr>
<tr><th><label for="id_form-3-name">Name:</label></th><td><input id="id_form-3-name" type="text" name="form-3-name" maxlength="100" /><input type="hidden" name="form-3-id" id="id_form-3-id" /></td></tr>
```

## *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 overide 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.forms %}
        {{ 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.forms %}
        {% 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.forms %}
        {{ 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`.

## 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.
    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.

# 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.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
```

As you can see it only displayed one form. This is because by default the `formset_factory` defines one extra form. This can be controlled with the `extra` parameter:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
```

## 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.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" value="Django is now open source" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub_date" value="2008-05-12" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub_date" id="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
```

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 force the maximum number of forms the formset will display:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
>>> formset = ArticleFormset()
>>> for form in formset.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
```

A `max_num` value of `0` (the default) puts no limit on the number 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)
>>> formset = ArticleFormSet({})
>>> 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-0-title': u'Test',
...     'form-0-pub_date': u'16 June 1904',
...     '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.

## Understanding the ManagementForm

You may have noticed the additional data that was required in the formset's data above. This data is coming from the `ManagementForm`. This form is dealt with internally to the formset. If you don't use it, it will result in an exception:

```
>>> data = {
...     'form-0-title': u'Test',
...     'form-0-pub_date': u'',
... }
>>> formset = ArticleFormSet(data)
...
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.

New in version 1.1: *Please, see the release notes*

## `total_form_count` and `initial_form_count`

`BaseModelFormSet` 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.

## 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-0-title': u'Test',
...     'form-0-pub_date': u'16 June 1904',
...     'form-1-title': u'Test',
...     'form-1-pub_date': u'23 June 1912',
... }
>>> 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 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.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" value="Article #1" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub_date" value="2008-05-10" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-ORDER">Order:</label></th><td><input type="text" name="form-0-ORDER" value="1" id="id_form-0-ORDER" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" value="Article #2" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub_date" value="2008-05-11" id="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-1-ORDER">Order:</label></th><td><input type="text" name="form-1-ORDER" value="2" id="id_form-1-ORDER" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
<tr><th><label for="id_form-2-ORDER">Order:</label></th><td><input type="text" name="form-2-ORDER" id="id_form-2-ORDER" /></td></tr>
```

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. Lets look at what will happen when the user changes these values:

```
>>> data = {
...     'form-TOTAL_FORMS': u'3',
...     'form-INITIAL_FORMS': u'2',
...     '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 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.forms:
....    print form.as_table()
<input type="hidden" name="form-TOTAL_FORMS" value="3" id="id_form-TOTAL_FORMS" /><input type="hidden" name="form-INITIAL_FORMS" value="2" id="id_form-INITIAL_FORMS" />
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" value="Article #1" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub_date" value="2008-05-10" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-DELETE">Delete:</label></th><td><input type="checkbox" name="form-0-DELETE" id="id_form-0-DELETE" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text" name="form-1-title" value="Article #2" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text" name="form-1-pub_date" value="2008-05-11" id="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-1-DELETE">Delete:</label></th><td><input type="checkbox" name="form-1-DELETE" id="id_form-1-DELETE" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text" name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text" name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
<tr><th><label for="id_form-2-DELETE">Delete:</label></th><td><input type="checkbox" name="form-2-DELETE" id="id_form-2-DELETE" /></td></tr>
```

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-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.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text" name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-my_field">My field:</label></th><td><input type="text" name="form-0-my_field" id="id_form-0-my_field" /></td></tr>
```

## 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
    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.forms %}
        {{ 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.

## 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.
    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.

## 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 model 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

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 `settings.MEDIA_URL`. For example, if the MEDIA_URL for your site was `http://media.example.com/`:

```
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://media.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 *form API reference*.

# Creating forms from models

## 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 |
| 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 |
| 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 with verify_exists set to the model field's verify_exists |
| XMLField | CharField with widget=forms.Textarea |

New in version 1.0: The FloatField form field and DecimalField model and form fields are new in Django 1.0.

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 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.):

```
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 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`.

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.

   Changed in version 1.1: *Please, see the release notes*

   The form will render the fields in the same order they are specified in the `fields` attribute.
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. 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

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 for a given model field. You 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 widget, then specify the `widget` parameter when declaring the form field:

```
>>> class ArticleForm(ModelForm):
...     pub_date = DateField(widget=MyDateWidget())
```

```
...
...     class Meta:
...         model = Article
```

## Changing the order of fields

New in version 1.1: *Please, see the release notes*

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.

## 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" name="form-INITIAL_FORMS" value="0" id="id_form-INITIAL_FORMS" />
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name" type="text" name="form-0-name" maxlength="100" /></td></tr>
<tr><th><label for="id_form-0-title">Title:</label></th><td><select name="form-0-title" id="id_form-0-title">
<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-birth_date" id="id_form-0-birth_date" /><input type="hidden" name="form-0-id" id="id_form-0-id" /></td></tr>
```

### 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):
        self.queryset = Author.objects.filter(name__startswith='O')
        super(BaseAuthorFormSet, self).__init__(*args, **kwargs)
```

Then, pass your `BaseAuthorFormSet` class to the factory function:

```
>>> AuthorFormSet = modelformset_factory(Author, formset=BaseAuthorFormSet)
```

### 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',))
```

### 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).

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` parameter to `modelformset_factory` to limit the number of forms displayed. With model formsets, this property limits the query to select only the maximum number of objects needed:

```
>>> Author.objects.order_by('name')
[<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]

>>> AuthorFormSet = modelformset_factory(Author, max_num=2, extra=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> formset.initial
[{'id': 1, 'name': u'Charles Baudelaire'}, {'id': 3, 'name': u'Paul Verlaine'}]
```

If the value of `max_num` is higher than the number of objects returned, 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.forms:
...     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="form-0-name" value="Charles Baudelaire" maxlength="100" /><input type="hidden" name="form-0-id" value="1" id="id_form-0-id" /></td></tr>
<tr><th><label for="id_form-1-name">Name:</label></th><td><input id="id_form-1-name" type="text" name="form-1-name" value="Paul Verlaine" maxlength="100" /><input type="hidden" name="form-1-id" value="3" id="id_form-1-id" /></td></tr>
<tr><th><label for="id_form-2-name">Name:</label></th><td><input id="id_form-2-name" type="text" name="form-2-name" value="Walt Whitman" maxlength="100" /><input type="hidden" name="form-2-id" value="2" id="id_form-2-id" /></td></tr>
<tr><th><label for="id_form-3-name">Name:</label></th><td><input id="id_form-3-name" type="text" name="form-3-name" maxlength="100" /><input type="hidden" name="form-3-id" id="id_form-3-id" /></td></tr>
```

## *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 overide 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.forms %}
        {{ 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.forms %}
        {% 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.forms %}
        {{ 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`.

### 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.
    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.

# 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).

## 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 e-mails, 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!

## Variables

Variables look like this: `{{ variable }}`. When the template engine encounters a variable, it evaluates that variable and replaces it with the result.

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

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.

See Using the built-in reference, below, for help on finding what variables are available in a given template.

## 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*.

## 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.

**ifequal and ifnotequal**
Display some contents if two arguments are or are not equal. For example:

```
{% ifequal athlete.name coach.name %}
    ...
{% endifequal %}
```

Or:

```
{% ifnotequal athlete.name "Joe" %}
    ...
{% endifnotequal %}
```

**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*.

## 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.

## 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 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" 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 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" 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.

## Automatic HTML escaping

New in version 1.0: *Please, see the release notes*

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 `'`
- `"` (double quote) is converted to `"`
- & 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 e-mail 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.

## Using the built-in reference

Django's admin interface includes a complete reference of all template tags and filters available for a given site. To see it, go to your admin interface and click the "Documentation" link in the upper right of the page.

The reference is divided into 4 sections: tags, filters, models, and views.

The **tags** and **filters** sections describe all the built-in tags (in fact, the tag and filter references below come directly from those pages) as well as any custom tag or filter libraries available.

The **views** page is the most valuable. Each URL in your site has a separate entry here, and clicking on a URL will show you:

- The name of the view function that generates that view.
- 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.

Each view documentation page also has a bookmarklet that you can use to jump from any page to the documentation page for that view.

Because Django-powered sites usually use database objects, the **models** section of the documentation page describes each type of object in the system along with all the fields available on that object.

Taken together, the documentation pages should tell you every tag, filter, variable and object available to you in a given template.

## 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.

# Generic views

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:

- Perform common "simple" tasks: redirect to a different page and render a given template.
- Display list and detail pages for a single object. If we were creating an application to manage conferences then a `talk_list` view and a `registered_user_list` view 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. The Django Weblog's (http://www.djangoproject.com/weblog/) year, month, and day archives are built with these, as would be a typical newspaper's archives.
- 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.

## Using generic views

All of these views are used by creating configuration dictionaries in your URLconf files and passing those dictionaries as the third member of the URLconf tuple for a given pattern.

For example, here's a simple URLconf you could use to present a static "about" page:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    })
)
```

Though this might seem a bit "magical" at first glance -- look, a view with no code! --, actually the `direct_to_template` view simply grabs information from the extra-parameters dictionary and uses that information when rendering the view.

Because this generic view -- and all the others -- is a regular view functions like any other, we can reuse it inside our own views. As an example, let's extend our "about" example to map URLs of the form `/about/<whatever>/` to statically rendered `about/<whatever>.html`. We'll do this by first modifying the URLconf to point to a view function:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
from mysite.books.views import about_pages

urlpatterns = patterns('',
```

```
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    ('^about/(w+)/$', about_pages),
)
```

Next, we'll write the about_pages view:

```
from django.http import Http404
from django.template import TemplateDoesNotExist
from django.views.generic.simple import direct_to_template

def about_pages(request, page):
    try:
        return direct_to_template(request, template="about/%s.html" % page)
    except TemplateDoesNotExist:
        raise Http404()
```

Here we're treating `direct_to_template` like any other function. Since it returns an `HttpResponse`, we can simply return it as-is. The only slightly tricky business here is dealing with missing templates. We don't want a nonexistent template to cause a server error, so we catch `TemplateDoesNotExist` exceptions and return 404 errors instead.

### Is there a security vulnerability here?

Sharp-eyed readers may have noticed a possible security hole: we're constructing the template name using interpolated content from the browser (`template="about/%s.html" % page`). At first glance, this looks like a classic *directory traversal* vulnerability. But is it really?

Not exactly. Yes, a maliciously crafted value of page could cause directory traversal, but although page *is* taken from the request URL, not every value will be accepted. The key is in the URLconf: we're using the regular expression `\w+` to match the page part of the URL, and `\w` only accepts letters and numbers. Thus, any malicious characters (dots and slashes, here) will be rejected by the URL resolver before they reach the view itself.

## Generic views of objects

The `direct_to_template` certainly is useful, but Django's generic views really shine when it comes to presenting views on 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 take a look at one of these generic views: the "object list" view. 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()

    def __unicode__(self):
        return self.name

    class Meta:
        ordering = ["-name"]

class Book(models.Model):
    title = models.CharField(max_length=100)
```

```
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

To build a list page of all books, we'd use a URLconf along these lines:

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    "queryset" : Publisher.objects.all(),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

That's all the Python code we need to write. We still need to write a template, however. We could explicitly tell the `object_list` view which template to use by including a `template_name` key in the extra arguments dictionary, 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.

This template will be rendered against a context containing a variable called `object_list` that contains all the book 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 "info" dictionary passed to the generic view. The *generic views reference* documents all the generic views and all their options in detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

## 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.

Luckily, in nearly every one of these cases, there are ways to simply extend generic views to handle a larger array of use cases. These situations usually fall into a handful of patterns dealt with in the sections that follow.

### Making "friendly" template contexts

You might have noticed that our sample publisher list template stores all the books 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 books here. A better name for that variable would be `publisher_list`; that variable's content is pretty obvious.

We can change the name of that variable easily with the `template_object_name` argument:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
}
```

```
urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

Providing a useful `template_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 other publishers on each publisher detail page. The `object_detail` generic view provides the publisher to the context, but it seems there's no way to get a list of *all* publishers in that template.

But there is: all generic views take an extra optional parameter, `extra_context`. This is a dictionary of extra objects that will be added to the template's context. So, to provide the list of all publishers on the detail detail view, we'd use an info dict like this:

```
from mysite.books.models import Publisher, Book

publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : Book.objects.all()}
}
```

This would populate a `{{ book_list }}` variable in the template context. This pattern can be used to pass any information down into the template for the generic view. It's very handy.

However, there's actually a subtle bug here -- can you spot it?

The problem has to do with when the queries in `extra_context` are evaluated. Because this example puts `Publisher.objects.all()` in the URLconf, it will be evaluated only once (when the URLconf is first loaded). Once you add or remove publishers, you'll notice that the generic view doesn't reflect those changes until you reload the Web server (see *Caching and QuerySets* for more information about when QuerySets are cached and evaluated).

### Note

This problem doesn't apply to the `queryset` generic view argument. Since Django knows that particular QuerySet should *never* be cached, the generic view takes care of clearing the cache when each view is rendered.

The solution is to use a callback in `extra_context` instead of a value. Any callable (i.e., a function) that's passed to `extra_context` will be evaluated when the view is rendered (instead of only once). You could do this with an explicitly defined function:

```
def get_books():
    return Book.objects.all()

publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : get_books}
}
```

or you could use a less obvious but shorter version that relies on the fact that `Book.objects.all` is itself a callable:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : Book.objects.all}
}
```

Notice the lack of parentheses after `Book.objects.all`; this references the function without actually calling it (which the generic view will do later).

## *Viewing subsets of objects*

Now let's take a closer look at this `queryset` key we've been using all along. Most generic views take one of these `queryset` arguments -- it's how the view knows which set of objects to display (see *Making queries* for more information about `QuerySet` objects, and see the *generic 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:

```
book_info = {
    "queryset" : Book.objects.all().order_by("-publication_date"),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/$', list_detail.object_list, book_info),
)
```

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:

```
acme_books = {
    "queryset": Book.objects.filter(publisher__name="Acme Publishing"),
    "template_name" : "books/acme_list.html"
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/acme/$', list_detail.object_list, acme_books),
)
```

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 *generic views reference* for more details.

## *Complex filtering with wrapper functions*

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? We can "wrap" the `object_list` generic view to avoid writing a lot of code by hand. As usual, we'll start by writing a URLconf:

```
from mysite.books.views import books_by_publisher

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/(w+)/$', books_by_publisher),
)
```

Next, we'll write the `books_by_publisher` view itself:

```
from django.http import Http404
from django.views.generic import list_detail
from mysite.books.models import Book, Publisher

def books_by_publisher(request, name):
```

```
    # Look up the publisher (and raise a 404 if it can't be found).
    try:
        publisher = Publisher.objects.get(name__iexact=name)
    except Publisher.DoesNotExist:
        raise Http404

    # Use the object_list view for the heavy lifting.
    return list_detail.object_list(
        request,
        queryset = Book.objects.filter(publisher=publisher),
        template_name = "books/books_by_publisher.html",
        template_object_name = "books",
        extra_context = {"publisher" : publisher}
    )
```

This works because there's really nothing special about generic views -- they're just Python functions. Like any view function, generic views expect a certain set of arguments and return `HttpResponse` objects. Thus, it's incredibly easy to wrap a small function around a generic view that does additional work before (or after; see the next section) handing things off to the generic view.

> **Note**
>
> Notice that in the preceding example we passed the current publisher being displayed in the `extra_context`. This is usually a good idea in wrappers of this nature; it lets the template know which "parent" object is currently being browsed.

## 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)
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')
    last_accessed = models.DateTimeField()
```

The generic `object_detail` view, 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 mysite.books.views import author_detail

urlpatterns = patterns('',
    #...
    (r'^authors/(?P<author_id>d+)/$', author_detail),
)
```

Then we'd write our wrapper function:

```
import datetime
from mysite.books.models import Author
from django.views.generic import list_detail
from django.shortcuts import get_object_or_404
```

```
def author_detail(request, author_id):
    # Look up the Author (and raise a 404 if she's not found)
    author = get_object_or_404(Author, pk=author_id)

    # Record the last accessed date
    author.last_accessed = datetime.datetime.now()
    author.save()

    # Show the detail page
    return list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )
```

### Note

This code won't actually work unless you create a `books/author_detail.html` template.

We can use a similar idiom to alter the response returned by the generic view. If we wanted to provide a downloadable plain-text version of the list of authors, we could use a view like this:

```
def author_list_plaintext(request):
    response = list_detail.object_list(
        request,
        queryset = Author.objects.all(),
        mimetype = "text/plain",
        template_name = "books/author_list.txt"
    )
    response["Content-Disposition"] = "attachment; filename=authors.txt"
    return response
```

This works because the generic views return simple `HttpResponse` objects that can be treated like dictionaries to set HTTP headers. This `Content-Disposition` business, by the way, instructs the browser to download and save the page instead of displaying it in the browser.

# Managing files

New in version 1.0: *Please, see the release notes*

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.

## 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.

## The File object

Internally, Django uses a `django.core.files.File` 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 `File` attributes and methods documented in *The File object*.

## 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`.

# 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.

## 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:

- **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]
  ```

- **Unit tests** -- tests that are expressed as methods on a Python class that subclasses `unittest.TestCase`. For example:

```
import unittest

class MyFuncTestCase(unittest.TestCase):
    def testBasic(self):
        a = ['larry', 'curly', 'moe']
        self.assertEquals(my_func(a, 0), 'larry')
        self.assertEquals(my_func(a, 1), 'curly')
```

You can choose the test framework you like, depending on which syntax you prefer, or you can mix and match, using one framework for some of your code and the other framework for other code. You can also use any *other* Python test frameworks, as we'll explain in a bit.

## *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.

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.

Here is an example model doctest:

```
# 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 how doctest works, see the standard library documentation for doctest.

## Writing unit tests

Like doctests, Django's unit tests use a standard library module: unittest. This module uses a different way of defining tests, taking a class-based approach.

As with doctests, 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.

This example `unittest.TestCase` subclass is equivalent to the example given in the doctest section above:

```
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 testSpeaking(self):
        self.assertEquals(self.lion.speak(), 'The lion says "roar"')
        self.assertEquals(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 standard library unittest 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!).

  If you're just getting started with testing, using doctests will probably get you started faster.
- 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.

Again, remember that you can use both systems side-by-side (even in the same app). In the end, most projects will eventually end up using both. Each shines in different circumstances.

## Running tests

Once you've written tests, run them using 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`.

New in version 1.0: You can now choose which test to run.

If you use unit tests, as opposed to doctests, you can be even *more* specific in choosing which tests to execute. 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.testFluffyAnimals
```

## The test database

Tests that require a database (namely, model tests) will not use your "real" (production) database. A separate, blank database is created for the tests.

Regardless of whether the tests pass or fail, the test database is destroyed when all the tests have been executed.

By default this test database gets its name by prepending `test_` to the value of the `DATABASE_NAME` setting. 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 the `TEST_DATABASE_NAME` setting.

Aside from using a separate database, the test runner will otherwise use all of the same database settings you have in your settings file: `DATABASE_ENGINE`, `DATABASE_USER`, `DATABASE_HOST`, etc. The test database is created by the user specified by `DATABASE_USER`, so you'll need to make sure that the given user account has sufficient privileges to create a new database on the system.

New in version 1.0: *Please, see the release notes*

For fine-grained control over the character encoding of your test database, use the `TEST_DATABASE_CHARSET` setting. If you're using MySQL, you can also use the `TEST_DATABASE_COLLATION` setting to control the particular collation used by the test database. See the *settings documentation* for details of these advanced settings.

## 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 the total 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.

## 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 Twill, 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 such as Twill and Selenium to test *rendered* HTML and the *behavior* of Web pages, namely JavaScript functionality.

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 PUBLIC "-//W3C//DTD XHTML 1.0 ...'
```

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.

## Making requests

Use the `django.test.client.Client` class to make requests. It requires no arguments at time of construction:

*class* **`Client`**

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.

New in version 1.1: *Please, see the release notes*

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 URL 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()
>>> f = open('wishlist.doc')
>>> c.post('/customers/wishes/', {'name': 'fred', 'attachment': f})
>>> f.close()
```

(The name `attachment` here is not relevant; use whatever name your file-processing code expects.)

Note that you should manually close the file after it has been provided to `post()`.

The `extra` argument acts the same as for `Client.get()`.

Changed in version 1.1: *Please, see the release notes*

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/?vistor=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)`**

New in version 1.1: *Please, see the release notes*

Makes a HEAD request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces. Acts just like `Client.get()` except it does not return a message body.

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.

**`options (path, data={}, follow=False, **extra)`**

New in version 1.1: *Please, see the release notes*

Makes an OPTIONS request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

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.

The `extra` argument acts the same as for `Client.get()`.

**`put (path, data={}, content_type=MULTIPART_CONTENT, follow=False, **extra)`**

New in version 1.1: *Please, see the release notes*

Makes an PUT request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces. Acts just like `Client.post()` except with the PUT request method.

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.

**`delete (path, follow=False, **extra)`**

New in version 1.1: *Please, see the release notes*

Makes an DELETE request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

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.

The `extra` argument acts the same as for `Client.get()`.

**`login (**credentials)`**

New in version 1.0: *Please, see the release notes*

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 ()`**

New in version 1.0: *Please, see the release notes*

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.
New in version 1.1: *Please, see the release notes*
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 RFC2616 for a full list of HTTP status codes.

**`template`**
The `Template` instance that was used to render the final content. 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'`.)
If the rendered page used multiple templates -- e.g., using *template inheritance* -- then `template` will be a list of `Template` instances, in the order in which they were rendered.

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 `unittest.TestCase.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 Cookie module documentation for more.

`Client.`**`session`**
A dictionary-like object containing session information. See the *session documentation* for full details.

## *Example*

The following is a simple unit test using the test client:

```
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.failUnlessEqual(response.status_code, 200)

        # Check that the rendered context contains 5 customers.
        self.failUnlessEqual(len(response.context['customers']), 5)
```

## *TestCase*

Normal Python unit test classes extend a base class of `unittest.TestCase`. Django provides an extension of this base class:

*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.

New in version 1.1: *Please, see the release notes*

*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 before the test runs by truncating all tables and reloading initial data. 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 and reload initial data at the beginning of 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. In order to guarantee that all TestCase code starts with a clean database, the Django test runner runs all TestCase tests first, before any other tests (e.g. doctests) that may alter the database without restoring it to its original 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**
>
> The TestCase use of rollback to un-do the effects of the test code may reveal previously-undetected errors in test code. For example, test code that assumes primary keys values will be assigned starting at one may find that assumption no longer holds true when rollbacks instead of table truncation are being used to reset the database. Similarly, the reordering of tests so that all TestCase classes run first may reveal unexpected dependencies on test case ordering. In such cases a quick fix is to switch the TestCase to a TransactionTestCase. A better long-term fix, that allows the test to take advantage of the speed benefit of TestCase, is to fix the underlying test problem.

## Default test client

New in version 1.0: *Please, see the release notes*

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:

```
import unittest
from django.test.client import Client

class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get('/customer/details/')
        self.failUnlessEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get('/customer/index/')
        self.failUnlessEqual(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.failUnlessEqual(response.status_code, 200)

    def test_index(self):
        response = self.client.get('/customer/index/')
        self.failUnlessEqual(response.status_code, 200)
```

## 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.

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.

    def testFluffyAnimals(self):
        # A test that uses the fixtures.
```

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*

New in version 1.0: *Please, see the release notes*

`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``.
```

This test case will use the contents of `myapp.test_urls` as the URLconf for the duration of the test case.

## Emptying the test outbox

New in version 1.0: *Please, see the release notes*

If you use Django's custom `TestCase` class, the test runner will clear the contents of the test e-mail outbox at the start of each test case.

For more detail on e-mail services during tests, see E-mail services.

## Assertions

New in version 1.0: *Please, see the release notes*

As Python's normal `unittest.TestCase` class implements assertion methods such as `assertTrue` and `assertEquals`, Django's custom `TestCase` class provides a number of custom assertion methods that are useful for testing Web applications:

**`TestCase.assertContains (`*`response`*`, `*`text`*`, `*`count=None`*`, `*`status_code=200`*`)`**

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.

**`TestCase.assertNotContains (`*`response`*`, `*`text`*`, `*`status_code=200`*`)`**

Asserts that a `Response` instance produced the given `status_code` and that `text` does not appears in the content of the response.

**`TestCase.assertFormError (`*`response`*`, `*`form`*`, `*`field`*`, `*`errors`*`)`**

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`*`)`**

Asserts that the template with the given name was used in rendering the response.
The name is a string such as `'admin/index.html'`.

**`TestCase.assertTemplateNotUsed (`*`response`*`, `*`template_name`*`)`**

Asserts that the template with the given name was *not* used in rendering the response.

**`TestCase.assertRedirects (`*`response`*`, `*`expected_url`*`, `*`status_code=302`*`, `*`target_status_code=200`*`)`**

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`.
New in version 1.1: *Please, see the release notes*
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.

## E-mail services

New in version 1.0: *Please, see the release notes*

If any of your Django views send e-mail using *Django's e-mail functionality*, you probably don't want to send e-mail each time you run a test using that view. For this reason, Django's test runner automatically redirects all Django-sent e-mail to a dummy outbox. This lets you test every aspect of sending e-mail -- 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 `SMTPConnection` class with a different version. (Don't worry -- this has no effect on any other e-mail 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 e-mail is saved in `django.core.mail.outbox`. This is a simple list of all  instances that have been sent. It does not exist under normal execution conditions, i.e., when you're not running unit tests. The outbox is created during test setup, along with the dummy . When the test framework is torn down, the standard  class is restored, and the test outbox is destroyed.

The outbox attribute is a special attribute that is created *only* when the tests are run. 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.assertEquals(len(mail.outbox), 1)

        # Verify that the subject of the first message is correct.
        self.assertEquals(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 = []
```

## 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.run_tests'. This method defines the default Django testing behavior. This behavior involves:

1. Performing global pre-test setup.
2. Creating the test database.
3. Running syncdb to install models and initial data into the test database.
4. Looking for unit tests and doctests in the models.py and tests.py files in each installed application.
5. Running the unit tests and doctests that are found.
6. Destroying the test database.
7. Performing global post-test teardown.

If you define your own test runner method and point TEST_RUNNER at that method, 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.

## Defining a test runner

New in version 1.0: *Please, see the release notes*

By convention, a test runner should be called run_tests. The only strict requirement is that it has the same arguments as the Django test runner:

**run_tests** (*test_labels*, *verbosity=1*, *interactive=True*, *extra_tests=***[**, **]**)

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.

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.
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 module_list.
This method should return the number of tests that failed.

## 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 SMTPConnection.

**teardown_test_environment** ()
Performs any global post-test teardown, such as removing the black magic hooks into the template system and restoring normal e-mail 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 settings.DATABASE_NAME to match the name of the test database.
Changed in version 1.0: create_test_db() now returns the name of the test database.

**destroy_test_db** (*old_database_name*, *verbosity=1*)
Destroys the database whose name is in the DATABASE_NAME setting and restores the value of DATABASE_NAME to the provided name.
verbosity has the same behavior as in run_tests().

# 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.

## 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.
- Messages: A simple way to queue messages for given users.

## Installation

Authentication support is bundled as a Django application in django.contrib.auth. To install it, do the following:

1. Put 'django.contrib.auth' in your INSTALLED_APPS setting.
2. Run the command manage.py syncdb.

Note that the default settings.py file created by django-admin.py startproject includes 'django.contrib.auth' in INSTALLED_APPS for convenience. If your INSTALLED_APPS already contains 'django.contrib.auth', 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.

# Users

*class* models.**User**

## API reference

### Fields

*class* models.**User**

User objects have the following fields:

**username**
Required. 30 characters or fewer. Alphanumeric characters only (letters, digits and underscores).

**first_name**
Optional. 30 characters or fewer.

**last_name**
Optional. 30 characters or fewer.

**email**
Optional. E-mail 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. Set this flag to False instead of deleting accounts.
This doesn't control whether or not the user can log in. Nothing in the authentication path checks the is_active flag, so if you want to reject a login based on is_active being False, it is up to you to check that in your own login view. However, permission checking using the methods like has_perm() does check this flag and will always 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: models.User. 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 ()**

New in version 1.0: *Please, see the release notes*

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 ()**

New in version 1.0: *Please, see the release notes*

Returns False if set_unusable_password() has been called for this user.

**get_group_permissions ()**

Returns a list of permission strings that the user has, through his/her groups.

**get_all_permissions ()**

Returns a list of permission strings that the user has, both through group and user permissions.

**has_perm (*perm*)**

Returns True if the user has the specified permission, where perm is in the format "<application name>.<lowercased model name>". If the user is inactive, this method will always return False.

**has_perms (*perm_list*)**

Returns True if the user has each of the specified permissions, where each perm is in the format "package.codename". If the user is inactive, this method will always return False.

**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.

**get_and_delete_messages ()**
Returns a list of `Message` objects in the user's queue and deletes the messages from the queue.

**email_user (*subject*, *message*, *from_email=None*)**
Sends an e-mail 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. 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*, *password=None*)**
Creates, saves and returns a `User`. The `username`, `email` and `password` are set as given, and the `User` gets `is_active=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='abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ23456789'*)**
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

Change a password with `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.

## Passwords

The `password` attribute of a `User` object is a string in this format:

```
hashtype$salt$hash
```

That's hashtype, salt and hash, separated by the dollar-sign character.

Hashtype is either `sha1` (default), `md5` or `crypt` -- the algorithm used to perform a one-way hash of the password. Salt is a random string used to salt the raw password to create the hash. Note that the `crypt` method is only supported on platforms that have the standard Python `crypt` module available.

New in version 1.0: Support for the `crypt` module is new in Django 1.0.

For example:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

The `set_password()` and `check_password()` functions handle the setting and checking of these values behind the scenes.

Previous Django versions, such as 0.90, used simple MD5 hashes without password salts. For backwards compatibility, those are still supported; they'll be converted automatically to the new style the first time `check_password()` works correctly for a given user.

## 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`.
- `has_perm()` always returns `False`.
- `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

New in version 1.0: The `manage.py createsuperuser` command is new.

`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 ForeignKey from your model to the User model, specified with unique=True to ensure only one instance of your model can be created for each User.

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 the profile, if it does not exist. You need to register a handler for the signal django.db.models.signals.post_save on the User model, and, in the handler, if created=True, create the associated user profile.

For more information, see Chapter 12 of the Django book.

## 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.

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 checking a user's password

**check_password** ()

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.models.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.

## 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.

Changed in version 1.0: Calling `logout()` now cleans session data.

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()`.

# 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`** ()

As a shortcut, you can use the convenient `login_required()` decorator:

```
from django.contrib.auth.decorators import login_required

def my_view(request):
    # ...
my_view = login_required(my_view)
```

Here's an equivalent example, using the more compact decorator syntax introduced in Python 2.4:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    # ...
```

`login_required()` also takes an optional `redirect_field_name` parameter. Example:

```
from django.contrib.auth.decorators import login_required

def my_view(request):
    # ...
my_view = login_required(redirect_field_name='redirect_to')(my_view)
```

Again, an equivalent example of the more compact decorator syntax introduced in Python 2.4:

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='redirect_to')
def my_view(request):
    # ...
```

`login_required()` does the following:

- If the user isn't logged in, redirect to `settings.LOGIN_URL` (`/accounts/login/` by default), passing the current absolute URL in the query string as `next` or the value of `redirect_field_name`. For 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.

Note that 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*])

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 %}">
<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 %}
```

## 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`:

views.**logout** (*request*[, *next_page*, *template_name*, *redirect_field_name*])

Logs a user out.

**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. This will default 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.

`views.`**`logout_then_login`** (*request*[, *login_url*])

Logs a user out, then redirects to the login page.

**Optional arguments:**

- `login_url`: The URL of the login page to redirect to. This will default to `settings.LOGIN_URL` if not supplied.

`views.`**`password_change`** (*request*[, *template_name*, *post_change_redirect*])

Allows a user to change their password.

**Optional arguments:**

- `template_name`: The full name of a template to use for displaying the password change form. This will default to `registration/password_change_form.html` if not supplied.
- `post_change_redirect`: The URL to redirect to after a successful password change.

**Template context:**

- `form`: The password change form.

`views.`**`password_change_done`** (*request*[, *template_name*])

The page shown after a user has changed their password.

**Optional arguments:**

- `template_name`: The full name of a template to use. This will default to `registration/password_change_done.html` if not supplied.

`views.`**`password_reset`** (*request*[, *is_admin_site*, *template_name*, *email_template_name*, *password_reset_form*, *token_generator*, *post_reset_redirect*])

Allows a user to reset their password, and sends them the new password in an e-mail.

**Optional arguments:**

- `template_name`: The full name of a template to use for displaying the password reset form. This will default to `registration/password_reset_form.html` if not supplied.
- `email_template_name`: The full name of a template to use for generating the e-mail with the new password. This will default to `registration/password_reset_email.html` if not supplied.
- `password_reset_form`: Form that will be used to set the password. Defaults to `SetPasswordForm`.
- `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.

**Template context:**

- `form`: The form for resetting the user's password.

`views.`**`password_reset_done`** (*request*[, *template_name*])

The page shown after a user has reset their password.

**Optional arguments:**

- `template_name`: The full name of a template to use. This will default to `registration/password_reset_done.html` if not supplied.

`views.`**`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. This will default 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.

**`password_reset_confirm`** (*request*[, *uidb36*, *token*, *template_name*, *token_generator*, *set_password_form*, *post_reset_redirect*])

Presents a form for entering a new password.

**Optional arguments:**

- `uidb36`: The user's id encoded in base 36. This will default to `None`.
- `token`: Token to check that the password is valid. This will default 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. This will default to `SetPasswordForm`.
- `post_reset_redirect`: URL to redirect after the password reset done. This will default to `None`.

**password_reset_complete** (*request*[, *template_name*])

Presents a view which informs the user that the password has been successfully changed.

**Optional arguments:**

- `template_name`: The full name of a template to display the view. This will default to `registration/password_reset_complete.html`.

## 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 resetting a user's password and e-mailing the new password to them.

*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.is_authenticated() and request.user.has_perm('polls.can_vote')):
        return HttpResponse("You can't vote in this poll.")
    # ...
```

`decorators.`**user_passes_test** ()

As a shortcut, you can use the convenient `user_passes_test` decorator:

```
from django.contrib.auth.decorators import user_passes_test

def my_view(request):
    # ...
my_view = user_passes_test(lambda u: u.has_perm('polls.can_vote'))(my_view)
```

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.
Here's the same thing, using Python 2.4's decorator syntax:

```
from django.contrib.auth.decorators import user_passes_test

@user_passes_test(lambda u: u.has_perm('polls.can_vote'))
def my_view(request):
    # ...
```

`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).
Example in Python 2.3 syntax:

```
from django.contrib.auth.decorators import user_passes_test

def my_view(request):
    # ...
my_view = user_passes_test(lambda u: u.has_perm('polls.can_vote'), login_url='/login/')(my_view)
```

Example in Python 2.4 syntax:

```
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*

`decorators.`**`permission_required`** ()

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

def my_view(request):
    # ...
my_view = permission_required('polls.can_vote')(my_view)
```

As for the `User.has_perm()` method, permission names take the form "`<application name>.<lowercased model name>`" (i.e. `polls.choice` for a `Choice` 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

def my_view(request):
    # ...
my_view = permission_required('polls.can_vote', login_url='/loginpage/')(my_view)
```

As in the `login_required()` decorator, `login_url` defaults to `settings.LOGIN_URL`.

## *Limiting access to generic views*

To limit access to a *generic view*, write a thin wrapper around the view, and point your URLconf to your wrapper instead of the generic view itself. For example:

```
from django.views.generic.date_based import object_detail

@login_required
def limited_object_detail(*args, **kwargs):
    return object_detail(*args, **kwargs)
```

# 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`.

## Custom permissions

To create custom permissions for a given model object, use the `permissions` *model Meta attribute*.

This example model creates three custom permissions:

```
class USCitizen(models.Model):
    # ...
    class Meta:
        permissions = (
            ("can_drive", "Can drive"),
            ("can_vote", "Can vote in elections"),
            ("can_drink", "Can drink alcohol"),
        )
```

The only thing this does is create those extra permissions when you run `manage.py syncdb`.

## API reference

*class* `models.`**`Permission`**

Just like users, permissions are implemented in a Django model that lives in django/contrib/auth/models.py.

## Fields

`Permission` objects have the following fields:

`models.Permission.`**`name`**

Required. 50 characters or fewer. Example: `'Can vote'`.

`models.Permission.`**`content_type`**

Required. A reference to the `django_content_type` database table, which contains a record for each installed Django model.

`models.Permission.`**`codename`**

Required. 100 characters or fewer. Example: `'can_vote'`.

### Methods

Permission objects have the standard data-access methods like any other *Django model*.

## 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.core.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.core.context_processors.PermWrapper, which is a template-friendly proxy of permissions.

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 %}
```

### 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 e-mail messages.

## Messages

The message system is a lightweight way to queue messages for given users.

A message is associated with a `User`. There's no concept of expiration or timestamps.

Messages are used by the Django admin after successful actions. For example, `"The poll Foo was created successfully."` is a message.

The API is simple:

**`models.User.message_set.create`** **(*message*)**

To create a new message, use `user_obj.message_set.create(message='message_text')`.

To retrieve/delete messages, use `user_obj.get_and_delete_messages()`, which returns a list of `Message` objects in the user's queue (if any) and deletes the messages from the queue.

In this example view, the system saves a message for the user after creating a playlist:

```
def create_playlist(request, songs):
    # Create the playlist with the given songs.
    # ...
    request.user.message_set.create(message="Your playlist was added successfully.")
    return render_to_response("playlists/create.html",
        context_instance=RequestContext(request))
```

When you use `RequestContext`, the currently logged-in user and his/her messages are made available in the *template context* as the template variable `{{ messages }}`. Here's an example of template code that displays messages:

```
{% if messages %}
<ul>
    {% for message in messages %}
    <li>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

Note that `RequestContext` calls `get_and_delete_messages()` behind the scenes, so any messages will be deleted even if you don't display them.

Finally, note that this messages framework only works with users in the user database. To send messages to anonymous users, use the *session framework*.

## 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 subsequent authentication attempts for that user. This effectively means that authentication sources are cached, 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:
    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:
    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:
    """
    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:

    # ...

    def has_perm(self, user_obj, perm):
        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.

# 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 (http://www.squid-cache.org/) 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.

## 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 `CACHE_BACKEND` setting in your settings file. Here's an explanation of all available values for `CACHE_BACKEND`.

### 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's used by sites such as Facebook and Wikipedia to reduce database access and dramatically increase site performance.

Memcached is available for free at http://danga.com/memcached/ . It runs as a daemon and is allotted a specified amount of RAM. All it does is provide an 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 the Memcached Python bindings, which are not bundled with Django directly. Two versions of this are available. Choose and install *one* of the following modules:

- The fastest available option is a module called `cmemcache`, available at http://gijsbert.org/cmemcache/ .
- If you can't install `cmemcache`, you can install `python-memcached`, available at ftp://ftp.tummy.com/pub/python-memcached/ . If that URL is no longer valid, just go to the Memcached Web site (http://www.danga.com/memcached/) and get the Python bindings from the "Client APIs" section.

New in version 1.0: The `cmemcache` option is new in 1.0. Previously, only `python-memcached` was supported.

To use Memcached with Django, set `CACHE_BACKEND` to `memcached://ip:port/`, where `ip` is the IP address of the Memcached daemon and `port` is the port on which Memcached is running.

In this example, Memcached is running on localhost (127.0.0.1) port 11211:

```
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

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 `CACHE_BACKEND`, separated by semicolons.

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:

```
CACHE_BACKEND = 'memcached://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):

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11212;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 `CACHE_BACKEND` setting to `"db://tablename"`, where `tablename` is the name of the database table. In this example, the cache table's name is `my_cache_table`:

```
CACHE_BACKEND = 'db://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.

## Filesystem caching

To store cached items on a filesystem, use the `"file://"` cache type for `CACHE_BACKEND`. For example, to store cached data in `/var/tmp/django_cache`, use this setting:

```
CACHE_BACKEND = 'file:///var/tmp/django_cache'
```

Note that there are three forward slashes toward the beginning of that example. The first two are for `file://`, and the third is the first character of the directory path, `/var/tmp/django_cache`. If you're on Windows, put the drive letter after the `file://`, like this:

```
file://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 `CACHE_BACKEND` to `"locmem:///"`. For example:

```
CACHE_BACKEND = 'locmem:///'
```

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 `CACHE_BACKEND` like so:

```
CACHE_BACKEND = 'dummy:///'
```

## Using a custom cache backend

New in version 1.0: *Please, see the release notes*

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 a Python import path as the scheme portion (the part before the initial colon) of the `CACHE_BACKEND` URI, like so:

```
CACHE_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_BACKEND arguments

Each cache backend may take arguments. They're given in query-string style on the `CACHE_BACKEND` 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).
- `max_entries`: For the `locmem`, `filesystem` and `database` backends, 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=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.

In this example, `timeout` is set to `60`:

```
CACHE_BACKEND = "memcached://127.0.0.1:11211/?timeout=60"
```

In this example, `timeout` is `30` and `max_entries` is `400`:

```
CACHE_BACKEND = "locmem:///?timeout=30&max_entries=400"
```

Invalid arguments are silently ignored, as are invalid values of known arguments.

## The per-site cache

Changed in version 1.0: (previous versions of Django only provided a single `CacheMiddleware` instead of the two pieces described below).

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_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 every page that doesn't have GET or POST parameters. 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 (include Django's admin interface). Note that if you use `CACHE_MIDDLEWARE_ANONYMOUS_ONLY`, you should make sure you've activated `AuthenticationMiddleware`.

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.

New in version 1.0: *Please, see the release notes*

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.

## The per-view cache

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

def my_view(request):
    ...

my_view = cache_page(my_view, 60 * 15)
```

Or, using Python 2.4's decorator syntax:

```
@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.

## 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(my_view, 60 * 15)),
)
```

If you take this approach, don't forget to import cache_page within your URLconf.

## Template fragment caching

New in version 1.0: *Please, see the release notes*

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.

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.

## 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 `CACHE_BACKEND` setting:

```
>>> from django.core.cache import cache
```

The basic interface is `set(key, value, timeout_seconds)` and `get(key)`:

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

The `timeout_seconds` argument is optional and defaults to the `timeout` argument in the `CACHE_BACKEND` setting (explained above).

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'
```

New in version 1.0: *Please, see the release notes*

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}
```

Finally, you can delete keys explicitly with `delete()`. This is an easy way of clearing the cache for a particular object:

```
>>> cache.delete('a')
```

New in version 1.1: *Please, see the release notes*

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.

## 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 e-mail 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.

## 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."

By default, Django's cache system creates its cache keys using the requested path (e.g., `"/stories/2005/jun/23/bank_robbed/"`). 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

# Python 2.3 syntax.
def my_view(request):
    # ...
my_view = vary_on_headers(my_view, 'User-Agent')

# Python 2.4+ decorator syntax.
@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.

## 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.

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_SETTINGS` 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):
    # ...
```

## *Other optimizations*

Django comes with a few other pieces of middleware that can help optimize your apps' 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 moderns browsers, saving bandwidth and transfer time.

## *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.

# Conditional View Processing

New in version 1.1: *Please, see the release notes*

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 there is no match with the ETag, 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.

## 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):
    ...
```

Of course, if you're using Python 2.3 or prefer not to use the decorator syntax, you can write the same code as follows, there is no difference:

```
def front_page(request, blog_id):
    ...
front_page = condition(last_modified_func=latest_entry)(front_page)
```

## 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.

## 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.

## 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.

# Sending e-mail

Although Python makes sending e-mail relatively easy via the smtplib library, Django provides a couple of light wrappers over it, to make sending e-mail extra quick.

The code lives in a single module: `django.core.mail`.

## 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 e-mail sent with `django.core.mail` will be set to the value of your `DEFAULT_CHARSET` setting.

## *send_mail()*

The simplest way to send e-mail is using the function `django.core.mail.send_mail()`. Here's its definition:

> **`send_mail`** (*`subject`*, *`message`*, *`from_email`*, *`recipient_list`*, *`fail_silently=False`*, *`auth_user=None`*, *`auth_password=None`*)

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 e-mail address. Each member of `recipient_list` will see the other recipients in the "To:" field of the e-mail 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.

## *send_mass_mail()*

`django.core.mail.send_mass_mail()` is intended to handle mass e-mailing. Here's the definition:

> **`send_mass_mail`** (*`datatuple`*, *`fail_silently=False`*, *`auth_user=None`*, *`auth_password=None`*)

`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 e-mail message. As in `send_mail()`, recipients in the same `recipient_list` will all see the other addresses in the e-mail messages' "To:" field.

## *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.

## *mail_admins()*

`django.core.mail.mail_admins()` is a shortcut for sending an e-mail to the site admins, as defined in the `ADMINS` setting. Here's the definition:

> **`mail_admins`** (*`subject`*, *`message`*, *`fail_silently=False`*)

`mail_admins()` prefixes the subject with the value of the `EMAIL_SUBJECT_PREFIX` setting, which is `"[Django] "` by default.

The "From:" header of the e-mail will be the value of the `SERVER_EMAIL` setting.

This method exists for convenience and readability.

## *mail_managers() function*

`django.core.mail.mail_managers()` is just like `mail_admins()`, except it sends an e-mail to the site managers, as defined in the `MANAGERS` setting. Here's the definition:

> **`mail_managers`** (*`subject`*, *`message`*, *`fail_silently=False`*)

## *Examples*

This sends a single e-mail 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 e-mail:

```
datatuple = (
    ('Subject', 'Message.', 'from@example.com', ['john@example.com']),
    ('Subject', 'Message.', 'from@example.com', ['jane@example.com']),
)
send_mass_mail(datatuple)
```

## Preventing header injection

Header injection is a security exploit in which an attacker inserts extra e-mail headers to control the "To:" and "From:" in e-mail messages that your scripts generate.

The Django e-mail 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 e-mail function (e.g. `send_mail()`) will raise `django.core.mail.BadHeaderError` (a subclass of `ValueError`) and, hence, will not send the e-mail. It's your responsibility to validate all data before passing it to the e-mail functions.

If a `message` contains headers at the start of the string, the headers will simply be printed as the first bit of the e-mail 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.')
```

## The EmailMessage and SMTPConnection classes

New in version 1.0: *Please, see the release notes*

Django's `send_mail()` and `send_mass_mail()` functions are actually thin wrappers that make use of the `EmailMessage` and `SMTPConnection` classes in `django.core.mail`. If you ever need to customize the way Django sends e-mail, you can subclass these two classes to suit your needs.

### Note

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 e-mail, you'll need to create `EmailMessage` instances directly.

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 e-mail messages and retain the original functions only for backwards compatibility.

In general, `EmailMessage` is responsible for creating the e-mail message itself. `SMTPConnection` is responsible for the network connection side of the operation. This means you can reuse the same connection (an `SMTPConnection` instance) for multiple messages.

## *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.

- `subject`: The subject line of the e-mail.
- `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 e-mail.
- `connection`: An `SMTPConnection` 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 e-mail message.

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, using either the connection that is specified in the `connection` attribute, or creating a new connection if none already exists. 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` 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 e-mail, `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 e-mail; the classic example is to send both text and HTML versions of a message. With Django's e-mail 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 e-mail. 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 e-mail, 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 it to the subtype. For example:

```
msg = EmailMessage(subject, html_content, from_email, [to])
msg.content_subtype = "html"  # Main content is now text/html
msg.send()
```

## SMTPConnection Objects

*class* **SMTPConnection**

The `SMTPConnection` class is initialized with the host, port, username and password for the SMTP server. If you don't specify one or more of those options, they are read from your settings file.

If you're sending lots of messages at once, the `send_messages()` method of the `SMTPConnection` class is useful. It takes a list of `EmailMessage` instances (or subclasses) and sends them over a single connection. For example, if you have a function called `get_notification_email()` that returns a list of `EmailMessage` objects representing some periodic e-mail you wish to send out, you could send this with:

```
connection = SMTPConnection()   # Use default settings for connection
messages = get_notification_email()
connection.send_messages(messages)
```

## Testing e-mail sending

The are times when you do not want Django to send e-mails at all. For example, while developing a website, you probably don't want to send out thousands of e-mails -- but you may want to validate that e-mails will be sent to the right people under the right conditions, and that those e-mails will contain the correct content.

The easiest way to test your project's use of e-mail is to use a "dumb" e-mail server that receives the e-mails 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 more entailed testing and processing of e-mails locally, see the Python documentation on the SMTP Server.

# Internationalization

Django has full support for internationalization of text in code and templates. Here's how it works.

## Overview

The goal of internationalization is to allow a single Web application to offer its content and functionality in multiple languages.

You, the Django developer, can accomplish this goal by adding 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."

Django takes care of using these hooks to translate Web apps, on the fly, according to users' language preferences.

Essentially, Django does two things:

- It lets developers and template authors specify which parts of their apps should be translatable.
- It uses these hooks to translate Web apps for particular users according to their language preferences.

## If you don't need internationalization in your app

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. If `USE_I18N` is set to `False`, 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.

## If you do need internationalization: three steps

1. Embed translation strings in your Python code and templates.
2. Get translations for those strings, in whichever languages you want to support.
3. Activate the locale middleware in your Django settings.

## Behind the scenes

Django's translation machinery uses the standard `gettext` module that comes with Python.

## 1. How to specify translation strings

Translation strings specify "This text should be translated." These strings can appear in your Python code and templates. It's your responsibility to mark translatable strings; the system can only translate strings it knows about.

### 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 placeholders (the month and the day) with their positions 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.

## 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.

## Lazy translation

Use the function `django.utils.translation.ugettext_lazy()` to translate strings lazily -- when the value is accessed rather than when the `ugettext_lazy()` function is called.

For example, to translate a model's `help_text`, 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'))
```

In this example, `ugettext_lazy()` stores 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 template rendering on the Django admin site.

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 verbose name `ugettext_lazy`, 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'))
```

Always use lazy translations in *Django models*. Field names and table names should be marked for translation (otherwise, they won't be translated in the admin interface). This means writing explicit `verbose_name` and `verbose_name_plural` options in the `Meta` class, though, rather than relying on Django's default determination of `verbose_name` and `verbose_name_plural` by looking at the model's class name:

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))
    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('mythings')
```

## *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 your need you 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 zero, 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 a a `format specification for argument 'name', as in 'msgstr[0]', doesn't exist in 'msgid'` error when running `django-admin.py compilemessages` or a `KeyError` Python exception at runtime.

## 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.

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 original text will be returned unchanged. 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.

It's not possible to mix a template variable inside a string within `{% trans %}`. If your translations require strings with variables (placeholders), use `{% blocktrans %}`:

```
{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}
```

To translate a template expression -- say, using template filters -- you need to bind the expression to a local variable for use within the translation block:

```
{% blocktrans with value|filter as myvar %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

If you need to bind more than one expression inside a `blocktrans` tag, separate the pieces with and:

```
{% blocktrans with book|title as book_t and author|title as author_t %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

To pluralize, specify both the singular and plural forms with the `{% plural %}` tag, which appears within `{% blocktrans %}` and `{% endblocktrans %}`. Example:

```
{% blocktrans count list|length as counter %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

When you use the pluralization feature and bind additional values to local variables apart from the counter value that selects the translated literal to be used, have in mind that the `blocktrans` construct is internally converted to an `ungettext` call. This means the same *notes regarding ungettext variables* apply.

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 *3. How Django discovers language preference*, below.)
- `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).

## Working with lazy translation objects

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 a couple of helper functions.

### 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(u'John Lennon')
instrument = ugettext_lazy(u'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).

### The allow_lazy() decorator

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.

## 2. How to create language files

Once you've tagged your strings for later translation, you need to write (or obtain) the language translations themselves. 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. See the relevant *LocaleMiddleware note* for more details.

### 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.

## A note to Django veterans

The old tool `bin/make-messages.py` has been moved to the command `django-admin.py makemessages` to provide consistency throughout Django.

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 three places:

- The root directory of your Django project.
- The root directory of your Django app.
- The root `django` directory (not a Subversion checkout, but the one that is linked-to via `$PYTHONPATH` or is located somewhere on that path). This is only relevant when you are creating a translation for Django itself, see *Submitting and maintaining translations*.

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` 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
```

When creating JavaScript translation catalogs you need to use the special 'djangojs' domain, **not** `-e js`.

## No gettext?

If you don't have the `gettext` utilities installed, `django-admin.py 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 `django-admin 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.

## A note to Django veterans

The old tool `bin/compile-messages.py` has been moved to the command `django-admin.py compilemessages` to provide consistency throughout Django.

## Working on Windows?

If you're using Windows and need to install the GNU gettext utilities so `django-admin compilemessages` works see gettext on Windows for more information.

## 3. 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 your language, 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.
- 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:

- First, it looks for a django_language key in the current user's session.
- Failing that, it looks for a cookie.

  Changed in version 1.0: *Please, see the release notes*

  In Django version 0.96 and before, the cookie's name is hard-coded to django_language. In Django 1,0, The cookie name 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 basic translations for that language. For example, Django uses technical message IDs to translate date formats and time formats -- so you will need at least those translations for the system to work correctly.

  A good starting point is to copy the English `.po` file and to translate at least the technical messages -- maybe the validation messages, too.

  Technical message IDs are easily recognized; they're all upper case. You don't translate the message ID as with other messages, you provide the correct local variant on the provided English value. For example, with `DATETIME_FORMAT` (or `DATE_FORMAT` or `TIME_FORMAT`), this would be the format string that you want to use in your language. The format is identical to the format strings used by the now template tag.

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`.

## *Using translations in your own projects*

Django looks for translations by following this algorithm:

- First, it looks for a `locale` directory in the application directory of the view that's being called. If it finds a translation for the selected language, the translation will be installed.
- Next, it looks for a `locale` directory in the project directory. If it finds a translation, the translation will be installed.
- Finally, it checks the Django-provided base translation in `django/conf/locale`.

This way, you can write applications that include their own translations, and you can override base translations in your project path. Or, you can just build a big project out of several apps and put all translations into one big project message file. The choice is yours.

### *Note*

If you're using manually configured settings, as described *Using settings without setting DJANGO_SETTINGS_MODULE*, the `locale` directory in the project directory will not be examined, since Django loses the ability to work out the location of the project directory. (Django normally uses the location of the settings file to determine this, and a settings file doesn't exist if you're manually configuring your settings.)

All message file repositories are structured the same way. They are:

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- All paths listed in `LOCALE_PATHS` in your settings file are searched in that order for `<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

To create message files, you use the same `django-admin.py makemessages` tool as with the Django message files. You only need to be in the right place -- in the directory where either the `conf/locale` (in case of the source tree) or the `locale/` (in case of app messages or project messages) directory are located. And you use the same `django-admin.py compilemessages` to produce the binary `django.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.

Application message files are a bit complicated to discover -- they need the `LocaleMiddleware`. If you don't use the middleware, only the Django message files and project message files will be processed.

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 translations could produce weird problems with makemessages: makemessages will traverse all directories below the current path and so might put message IDs into the project message file 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 on the project level will only translate strings that are connected to your explicit project and not strings that are distributed independently.

## The set_language redirect view

As a convenience, Django comes with a view, django.views.i18n.set_language, that sets a user's language preference and redirects 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">
<input name="next" type="hidden" value="/next/page/" />
<select name="language">
{% for lang in LANGUAGES %}
<option value="{{ lang.0 }}">{{ lang.1 }}</option>
{% endfor %}
</select>
<input type="submit" value="Go" />
</form>
```

## Translations and JavaScript

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

The main solution to these problems is the 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 the application, project or Django core, according to what you specify in either the info_dict or the URL.

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.

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.

## 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).

## Creating JavaScript translation catalogs

You create and update the translation catalogs the same way as the other

Django translation catalogs -- with the django-admin.py makemessages tool. The only difference is you need to provide a `-d djangojs` parameter, like this:

```
django-admin.py makemessages -d djangojs -l de
```

This would create or update the translation catalog for JavaScript for German. After updating translation catalogs, just run `django-admin.py compilemessages` the same way as you do with normal Django translation catalogs.

## Specialties of Django translation

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.

## 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 recomend using `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. Some version 0.14.4 binaries have been found to not support this command. 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".

# Pagination

Changed in version 1.0: Pagination facilities have been almost fully reworked.

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`.

## 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()
3
>>> 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)
...
EmptyPage: That page number is less than 1
>>> p.page(3)
...
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.

## 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, InvalidPage, EmptyPage

def listing(request):
    contact_list = Contacts.objects.all()
    paginator = Paginator(contact_list, 25) # Show 25 contacts per page

    # Make sure page request is an int. If not, deliver first page.
    try:
        page = int(request.GET.get('page', '1'))
    except ValueError:
```

```
        page = 1

    # If page request (9999) is out of range, deliver last page of results.
    try:
        contacts = paginator.page(page)
    except (EmptyPage, InvalidPage):
        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.object_list %}
    {# 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>
```

## 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 `object_list.__len__()`. 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]`.

## `InvalidPage` exceptions

The `page()` method raises `InvalidPage` 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:

**`PageNotAnInteger`**
Raised when `page()` is given a value that isn't an integer.

**`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`.

## Page objects

**`Page(object_list, number, paginator):`**

You usually won't construct Pages by hand -- you'll get them using `Paginator.page()`.

## 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. Note that this is "dumb" and will return the next page number regardless of whether a

subsequent page exists.

**Page.previous_page_number ()**
Returns the previous page number. Note that this is "dumb" and will return the previous page number regardless of whether a previous page exists.

**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.

# Serializing Django objects

Django's serialization framework provides a mechanism for "translating" Django objects into other formats. Usually these other formats will be text-based and used for sending Django objects 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.

## 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 objects, 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`):

```
out = open("file.xml", "w")
xml_serializer.serialize(SomeModel.objects.all(), stream=out)
```

### 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)
```

## 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`.

## 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 (using a version of simplejson bundled with Django). |
| python | Translates to and from "simple" Python objects (lists, dicts, strings, etc.). Not really all that useful on its own, but used as a base for other serializers. |
| 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)
```

The Django source code includes the simplejson module. However, if you're using Python 2.6 (which includes a builtin version of the module), Django will use the builtin `json` module automatically. If you have a system installed version that includes the C-based speedup extension, or your system version is more recent than the version shipped with Django (currently, 2.0.7), the system version will be used instead of the version included with Django.

Be aware that if you're serializing using that module directly, not all Django output can be passed unmodified to simplejson. In particular, *lazy translation objects* need a special encoder written for them. Something like this will work:

```
from django.utils.functional import Promise
from django.utils.encoding import force_unicode

class LazyEncoder(simplejson.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Promise):
            return force_unicode(obj)
        return super(LazyEncoder, self).default(obj)
```

# 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.

## 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.

## 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_python)

In your live server environment, you'll need to tell Apache/mod_python which settings file to use. Do that with `SetEnv`:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>
```

Read the *Django mod_python documentation* for more information.

## 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.

## 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.

## 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.

## 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.

## Available settings

For a full list of available settings, see the *settings reference*.

## 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.

## 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.

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.

## 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 is called.
- `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.

## Listening to signals

To receive a signal, you need to register a *receiver* function that gets called when the signal is sent. 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

Next, we'll need to connect our receiver to the signal:

```
from django.core.signals import request_finished

request_finished.connect(my_callback)
```

Now, our `my_callback` function will be called each time a request finishes.

> ## 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 recieving 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 myapp.models import MyModel

def my_handler(sender, **kwargs):
    ...

pre_save.connect(my_handler, sender=MyModel)
```

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.

## 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

**Signal.send (*sender*, ***kwargs*)**

To send a signal, call `Signal.send()`. 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)
        ...
```

# "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.

## 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. 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.

### Configuring Apache

To check against Django's authorization database from a Apache configuration file, you'll need to use mod_python's `PythonAuthenHandler` directive along with the standard `Auth*` and `Require` directives:

```
<Location /example/>
    AuthType Basic
```

```
    AuthName "example.com"
    Require valid-user

    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonAuthenHandler django.contrib.auth.handlers.modpython
</Location>
```

## Using the authentication handler with Apache 2.2

If you're using Apache 2.2, you'll need to take a couple extra steps.

You'll need to ensure that `mod_auth_basic` and `mod_authz_user` are loaded. These might be compiled statically into Apache, or you might need to use LoadModule to load them dynamically (as shown in the example at the bottom of this note).

You'll also need to insert configuration directives that prevent Apache from trying to use other authentication modules, as well as specifying the `AuthUserFile` directive and pointing it to `/dev/null`. Depending on which other authentication modules you have loaded, you might need one or more of the following directives:

```
AuthBasicAuthoritative Off
AuthDefaultAuthoritative Off
AuthzLDAPAuthoritative Off
AuthzDBMAuthoritative Off
AuthzDefaultAuthoritative Off
AuthzGroupFileAuthoritative Off
AuthzOwnerAuthoritative Off
AuthzUserAuthoritative Off
```

A complete configuration, with differences between Apache 2.0 and Apache 2.2 marked in bold, would look something like:

```
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule authz_user_module modules/mod_authz_user.so

...

<Location /example/>
    AuthType Basic
    AuthName "example.com"
    AuthUserFile /dev/null
    AuthBasicAuthoritative Off
    Require valid-user

    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonAuthenHandler django.contrib.auth.handlers.modpython
</Location>
```

By default, the authentication handler will limit access to the `/example/` location to users marked as staff members. You can use a set of PythonOption directives to modify this behavior:

| PythonOption | Explanation |
|---|---|
| `DjangoRequireStaffStatus` | If set to on only "staff" users (i.e. those with the `is_staff` flag set) will be allowed.<br><br>Defaults to on. |
| `DjangoRequireSuperuserStatus` | If set to on only superusers (i.e. those with the `is_superuser` flag set) will be allowed.<br><br>Defaults to `off`. |
| `DjangoPermissionName` | The name of a permission to require for access. See *custom permissions* for more information.<br><br>By default no specific permission will be required. |

Note that sometimes SetEnv doesn't play well in this mod_python configuration, for reasons unknown. If you're having problems getting mod_python to recognize your DJANGO_SETTINGS_MODULE, you can set it using PythonOption instead of SetEnv. Therefore, these two Apache directives are equivalent:

```
SetEnv DJANGO_SETTINGS_MODULE mysite.settings
PythonOption DJANGO_SETTINGS_MODULE mysite.settings
```

# 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.

## 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'
```

## RemoteUserBackend

*class* django.contrib.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.

# Writing custom django-admin commands

New in version 1.0: *Please, see the release notes*

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.

To do this, just add a management/commands directory to your application. Each Python module in that directory will be auto-discovered and registered as a command that can be executed as an action when you run manage.py:

```
blog/
    __init__.py
    models.py
    management/
        __init__.py
        commands/
            __init__.py
            explode.py
    views.py
```

In this example, the explode command will be made available to any project that includes the blog application in settings.INSTALLED_APPS.

The explode.py module has only one requirement -- it must define a class called Command that extends django.core.management.base.BaseCommand.

For more details on how to define your own commands, look at the code for the existing django-admin.py commands, in /django/core/management/commands.

# Writing custom model fields

New in version 1.0: *Please, see the release notes*

## 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):
    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.

## 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?

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.

## *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):
    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.

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`: Currently only used with the Oracle backend and only for index creation. 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

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:

*class* `django.db.models.`**`SubfieldBase`**

For example:

```
class HandField(models.Field):
    __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.

## 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

**`db_type (`*`self`*`)`**

Returns the database column data type for the `Field`, taking into account the current `DATABASE_ENGINE` setting.

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):
        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)
    gender = models.CharField(max_length=1)
    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 import the Django settings module and check the `DATABASE_ENGINE` setting. For example:

```
class MyDateField(models.Field):
    def db_type(self):
        from django.conf import settings
        if settings.DATABASE_ENGINE == '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 `DATABASE_ENGINE` 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):
        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):
        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

**`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)]
        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.

**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 database values*

**`get_db_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 can be used as a parameter in a query for the database backend.

For example:

```
class HandField(models.Field):
    # ...

    def get_db_prep_value(self, value):
        return ''.join([''.join(l) for l in (value.north,
                value.east, value.south, value.west)])
```

**`get_db_prep_save`** **(*self*, *value*)**

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*

**`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

**get_db_prep_lookup (*self*, *lookup_type*, *value*)**

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_db_prep_lookup()`. If you don't, `get_db_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_db_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_db_prep_value()`, or at least factor out some common pieces.

For example, the following code implements `get_db_prep_lookup` to limit the accepted lookup types to `exact` and `in`:

```
class HandField(models.Field):
    # ...

    def get_db_prep_lookup(self, lookup_type, value):
        # We only handle 'exact' and 'in'. All others are errors.
        if lookup_type == 'exact':
            return [self.get_db_prep_value(value)]
        elif lookup_type == 'in':
            return [self.get_db_prep_value(v) for v in value]
        else:
            raise TypeError('Lookup type %r not supported.' % lookup_type)
```

## Specifying the form field for a model 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

**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

**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_db_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.

## 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.

# Custom template tags and filters

## Introduction

Django's template system comes 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.

## 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.

## 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()
```

## 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.template.defaultfilters import stringfilter

@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).

## 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).

If you're using Python 2.4 or above, you can use `register.filter()` as a decorator instead:

```
@register.filter(name='cut')
@stringfilter
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
@stringfilter
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.

## *Filters and auto-escaping*

New in version 1.0: *Please, see the release notes*

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 put the `is_safe` attribute on your filter function and set it to `True`, like so:

```
@register.filter
def myfilter(value):
    return value
myfilter.is_safe = True
```

This attribute 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
def add_xx(value):
    return '%sxx' % value
add_xx.is_safe = True
```

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` defaults to `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` attribute to `True` on your function. (If you don't specify this attribute, it defaults to `False`). This attribute 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

   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)
   initial_letter_filter.needs_autoescape = True
   ```

   The `needs_autoescape` attribute on the filter function 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` attribute 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` attribute won't change anything either way.

## 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()[0]
    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

New in version 1.0: *Please, see the release notes*

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.

## 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, in Python 2.4 and above:

```
@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.split()[0]
    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])
```

Changed in version 1.0: Variable resolution has changed in the 1.0 release of Django. `template.resolve_variable()` has been deprecated in favor of a new `template.Variable` class.

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.

## Shortcut for simple tags

Many template tags take a number of arguments -- strings or a template variables -- and return a string after doing some processing based solely on the input argument 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 the 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)
```

In Python 2.4, the decorator syntax also works:

```
@register.simple_tag
def current_time(format_string):
    ...
```

A couple of 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.

When your template tag does not need access to the current context, writing a function to work with the input values and using the `simple_tag` helper is the easiest way to create a new tag.

## 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)
```

As always, Python 2.4 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.

## Setting a variable in the context

The above example 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>
```

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:

```
{% get_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`.

## 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 the standard `{% comment %}` tag is 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 ''
```

`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 %}` and `{% ifchanged %}`. They live in `django/template/defaulttags.py`.

# 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:

## _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.

## _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).

## 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.

### *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 underscores to the filename until it finds one that's available in the destination directory.

# *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:

## *How to use Django with Apache and mod_wsgi*

Deploying Django with Apache and mod_wsgi is the recommended way to get Django into production.

mod_wsgi is an Apache module which can be used to host any Python application which supports the Python WSGI interface, 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 `httpd.conf` file and add:

```
WSGIScriptAlias / /path/to/mysite/apache/django.wsgi
```

The first bit above is the url you want to be serving 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. This tells Apache to serve any request below the given URL using the WSGI application defined by that file.

Next we'll need to actually create this WSGI application, so create the file mentioned in the second part of `WSGIScriptAlias` and add:

```
import os
import sys

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'

import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

If your project is not on your PYTHONPATH by default you can add:

```
sys.path.append('/usr/local/django')
```

just above the final `import` line to place your project on the path. Remember to replace 'mysite.settings' with your correct settings file, and '/usr/local/django' with your own project's location.

### *Serving media files*

Django doesn't serve media 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 `/media/` URL space as a static file. All other URLs will be served using mod_wsgi:

```
Alias /robots.txt /usr/local/wsgi/static/robots.txt
Alias /favicon.ico /usr/local/wsgi/static/favicon.ico

AliasMatch /([^/]*\.css) /usr/local/wsgi/static/styles/$1

Alias /media/ /usr/local/wsgi/static/media/

<Directory /usr/local/wsgi/static>
Order deny,allow
Allow from all
</Directory>

WSGIScriptAlias / /usr/local/wsgi/scripts/django.wsgi

<Directory /usr/local/wsgi/scripts>
Order allow,deny
Allow from all
</Directory>
```

More details on configuring a mod_wsgi site to serve static files can be found in the mod_wsgi documentation on hosting static files.

## Details

For more details, see the mod_wsgi documentation on Django integration, which explains the above in more detail, and walks through all the various options you've got when deploying under mod_wsgi.

## How to use Django with Apache and mod_python

The mod_python module for Apache can be used to deploy Django to a production server, although it has been mostly superseded by the simpler *mod_wsgi deployment option*.

mod_python is similar to (and inspired by) 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.

Django requires Apache 2.x and mod_python 3.x, and you should use Apache's prefork MPM, as opposed to the worker MPM.

### See also

- Apache is a big, complex animal, and this document only scratches the surface of what Apache can do. If you need more advanced information about Apache, there's no better source than Apache's own official documentation
- You may also be interested in *How to use Django with FastCGI, SCGI, or AJP*.

## Basic configuration

To configure Django with mod_python, first make sure you have Apache installed, with the mod_python module activated.

Then edit your `httpd.conf` file and add the following:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonOption django.root /mysite
    PythonDebug On
</Location>
```

...and replace `mysite.settings` with the Python import path to your Django project's settings file.

This tells Apache: "Use mod_python for any URL at or under '/mysite/', using the Django mod_python handler." It passes the value of *DJANGO_SETTINGS_MODULE* so mod_python knows which settings to use.

New in version 1.0: The `PythonOption django.root ...` is new in this version.

Because mod_python does not know we are serving this site from underneath the `/mysite/` prefix, this value needs to be passed through to the mod_python handler in Django, via the `PythonOption django.root ...` line. The value set on that line (the last item) should match the string given in the `<Location ...>` directive. The effect of this is that Django will automatically strip the `/mysite` string from the front of any URLs before matching them against your URLconf patterns. If you later move your site to live under `/mysite2`, you will not have to change anything except the `django.root` option in the config file.

When using `django.root` you should make sure that what's left, after the prefix has been removed, begins with a slash. Your URLconf patterns that are expecting an initial slash will then work correctly. In the above example, since we want to send things like `/mysite/admin/` to `/admin/`, we need to remove the string `/mysite` from the beginning, so that is the `django.root` value. It would be an error to use `/mysite/` (with a trailing slash) in this case.

Note that we're using the `<Location>` directive, not the `<Directory>` directive. The latter is used for pointing at places on your filesystem, whereas `<Location>` points at places in the URL structure of a Web site. `<Directory>` would be meaningless here.

Also, if your Django project is not on the default `PYTHONPATH` for your computer, you'll have to tell mod_python where your project can be found:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonOption django.root /mysite
    PythonDebug On
    PythonPath "['/path/to/project'] + sys.path"
</Location>
```

The value you use for `PythonPath` should include the parent directories of all the modules you are going to import in your application. It should also include the parent directory of the *DJANGO_SETTINGS_MODULE* location. This is exactly the same situation as setting the Python path for interactive usage. Whenever you try to import something, Python will run through all the directories in `sys.path` in turn, from first to last, and try to import from each directory until one succeeds.

Make sure that your Python source files' permissions are set such that the Apache user (usually named apache or `httpd` on most systems) will have read access to the files.

An example might make this clearer. Suppose you have some applications under `/usr/local/django-apps/` (for example, `/usr/local/django-apps/weblog/` and so forth), your settings file is at `/var/www/mysite/settings.py` and you have specified *DJANGO_SETTINGS_MODULE* as in the above example. In this case, you would need to write your `PythonPath` directive as:

```
PythonPath "['/usr/local/django-apps/', '/var/www'] + sys.path"
```

With this path, `import weblog` and `import mysite.settings` will both work. If you had `import blogroll` in your code somewhere and `blogroll` lived under the `weblog/` directory, you would *also* need to add `/usr/local/django-apps/weblog/` to your `PythonPath`. Remember: the **parent directories** of anything you import directly must be on the Python path.

## *Note*

If you're using Windows, we still recommended that you use forward slashes in the pathnames, even though Windows normally uses the backslash character as its native separator. Apache knows how to convert from the forward slash format to the native format, so this approach is portable and easier to read. (It avoids tricky problems with having to double-escape backslashes.)

This is valid even on a Windows system:

```
PythonPath "['c:/path/to/project'] + sys.path"
```

You can also add directives such as `PythonAutoReload Off` for performance. See the mod_python documentation for a full list of options.

Note that you should set `PythonDebug Off` on a production server. If you leave `PythonDebug On`, your users would see ugly (and revealing) Python tracebacks if something goes wrong within mod_python.

Restart Apache, and any request to `/mysite/` or below will be served by Django. Note that Django's URLconfs won't trim the "/mysite/" -- they get passed the full URL.

When deploying Django sites on mod_python, you'll need to restart Apache each time you make changes to your Python code.

## Multiple Django installations on the same Apache

It's entirely possible to run multiple Django installations on the same Apache instance. Just use `VirtualHost` for that, like so:

```
NameVirtualHost *

<VirtualHost *>
    ServerName www.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</VirtualHost>

<VirtualHost *>
    ServerName www2.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
</VirtualHost>
```

If you need to put two Django installations within the same `VirtualHost` (or in different `VirtualHost` blocks that share the same server name), you'll need to take a special precaution to ensure mod_python's cache doesn't mess things up. Use the `PythonInterpreter` directive to give different <Location> directives separate interpreters:

```
<VirtualHost *>
    ServerName www.example.com
    # ...
    <Location "/something">
        SetEnv DJANGO_SETTINGS_MODULE mysite.settings
        PythonInterpreter mysite
    </Location>

    <Location "/otherthing">
        SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
        PythonInterpreter othersite
    </Location>
</VirtualHost>
```

The values of `PythonInterpreter` don't really matter, as long as they're different between the two `Location` blocks.

## Running a development server with mod_python

If you use mod_python for your development server, you can avoid the hassle of having to restart the server each time you make code changes. Just set `MaxRequestsPerChild 1` in your `httpd.conf` file to force Apache to reload everything for each request. But don't do that on a production server, or we'll revoke your Django privileges.

If you're the type of programmer who debugs using scattered `print` statements, note that `print` statements have no effect in mod_python; they don't appear in the Apache log, as one might expect. If you have the need to print debugging information in a mod_python setup, either do this:

```
assert False, the_value_i_want_to_see
```

Or add the debugging information to the template of your page.

## *Serving media files*

Django doesn't serve media 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, here's how you can turn off mod_python for a particular part of the site:

```
<Location "/media">
    SetHandler None
</Location>
```

Just change `Location` to the root URL of your media files. You can also use `<LocationMatch>` to match a regular expression.

This example sets up Django at the site root but explicitly disables Django for the `media` subdirectory and any URL that ends with `.jpg`, `.gif` or `.png`:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>

<Location "/media">
    SetHandler None
</Location>

<LocationMatch "\.(jpg|gif|png)$">
    SetHandler None
</LocationMatch>
```

## *Serving the admin files*

Note that the Django development server automagically serves admin media files, 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/media`) of the Django distribution.

Here are two recommended approaches:

1. Create a symbolic link to the admin media files from within your document root. This way, all of your Django-related files -- code **and** templates -- stay in one place, and you'll still be able to `svn update` your code to get the latest admin templates, if they change.
2. Or, copy the admin media files so that they live within your Apache document root.

## *Using "eggs" with mod_python*

If you installed Django from a Python egg or are using eggs in your Django project, some extra configuration is required. Create an extra file in your project (or somewhere else) that contains something like the following:

```
import os
os.environ['PYTHON_EGG_CACHE'] = '/some/directory'
```

Here, `/some/directory` is a directory that the Apache webserver process can write to. It will be used as the location for any unpacking of code the eggs need to do.

Then you have to tell mod_python to import this file before doing anything else. This is done using the PythonImport directive to mod_python. You need to ensure that you have specified the `PythonInterpreter` directive to mod_python as described above (you need to do this even if you aren't serving multiple installations in this case). Then add the `PythonImport` line in the main server configuration (i.e., outside the `Location` or `VirtualHost` sections). For example:

```
PythonInterpreter my_django
PythonImport /path/to/my/project/file.py my_django
```

Note that you can use an absolute path here (or a normal dotted import path), as described in the mod_python manual. We use an absolute path in the above example because if any Python path modifications are required to access your project, they will not have been done at the time the `PythonImport` line is processed.

## *Error handling*

When you use Apache/mod_python, errors will be caught by Django -- in other words, they won't propagate to the Apache level and won't appear in the Apache `error_log`.

The exception for this is if something is really wonky in your Django setup. In that case, you'll see an "Internal Server Error" page in your browser and the full Python traceback in your Apache `error_log` file. The `error_log` traceback is spread over multiple lines. (Yes, this is ugly and rather hard to read, but it's how mod_python does things.)

## *If you get a segmentation fault*

If Apache causes a segmentation fault, there are two probable causes, neither of which has to do with Django itself.

1. It may be because your Python code is importing the "pyexpat" module, which may conflict with the version embedded in Apache. For full information, see Expat Causing Apache Crash.
2. It may be because you're running mod_python and mod_php in the same Apache instance, with MySQL as your database backend. In some cases, this causes a known mod_python issue due to version conflicts in PHP and the Python MySQL backend. There's full information in the mod_python FAQ entry.

If you continue to have problems setting up mod_python, a good thing to do is get a barebones mod_python site working, without the Django framework. This is an easy way to isolate mod_python-specific problems. Getting mod_python Working details this procedure.

The next step should be to edit your test code and add an import of any Django-specific code you're using -- your views, your models, your URLconf, your RSS configuration, etc. Put these imports in your test handler function and access your test URL in a browser. If this causes a crash, you've confirmed it's the importing of Django code that causes the problem. Gradually reduce the set of imports until it stops crashing, so as to find the specific module that causes the problem. Drop down further into modules and look into their imports, as necessary.

## *If you get a UnicodeEncodeError*

If you're taking advantage of the internationalization features of Django (see *Internationalization*) 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 FastCGI, SCGI, or AJP*

Although the current preferred setup for running Django is *Apache with mod_wsgi*, many people use shared hosting, on which protocols such as FastCGI, SCGI or AJP are the only viable options. In some setups, these protocols may provide better performance than mod_wsgi.

> ### *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 mod_python, FastCGI allows code to stay in memory, allowing requests to be served with no startup time. Unlike mod_python (or mod_perl), a FastCGI process doesn't run inside the Web server process, but 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. For mod_python, for example, every Apache process gets its own Python interpreter, which uses up a considerable amount of RAM.
>
> 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
```

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).

## 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")
```

### 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 *modpython* 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 webserver, 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 webserver 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 discusses deployment and especially scaling in more detail.

## Error reporting via e-mail

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 email you details of those errors.

## Server errors

When DEBUG is False, Django will e-mail the users listed in the ADMIN 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.

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.

## 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 e-mail the users listed in the MANAGERS setting whenever your code raises a 404 and the request has a referer. (It doesn't bother to e-mail 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_ENDS and IGNORABLE_404_STARTS settings. Both should be a tuple of strings. For example:

```
IGNORABLE_404_ENDS = ('.php', '.cgi')
IGNORABLE_404_STARTS = ('/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 best way to disable this behavior is to set SEND_BROKEN_LINK_EMAILS to False.

### 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.

# 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.

## 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. Every time you run `loaddata` the data will be read from the fixture and re-loaded into the database. Note that 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.

### See also

Fixtures are also used by the *testing framework* to help set up a consistent test environment.

## 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`, `sqlreset`, `sqlall` and `reset` 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.

## 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 MySQL. 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 value of `DATABASE_ENGINE` in your settings file (e.g., `postgresql`, `mysql`).

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.postgresql.sql` and you're installing the app on PostgreSQL, Django will execute the contents of `sql/person.postgresql.sql` first, then `sql/person.sql`.

# Running Django on 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.

## Installing Jython

Django works with Jython versions 2.5b3 and higher. Download Jython at http://www.jython.org/.

## 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.

## 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.

## 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 website. Also, read the database backends documentation there.

## 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 consistancy, 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`.

# 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.

## 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 these settings in your *settings file*:

- `DATABASE_NAME`
- `DATABASE_ENGINE`
- `DATABASE_USER`
- `DATABASE_PASSWORD`
- `DATABASE_HOST`
- `DATABASE_PORT`

## 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.

## 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
```

## 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.

# 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.

## 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.

## Using the template system

Alternatively, you can use the *Django template system* to generate CSV. This is lower-level than using the convenient CSV, 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 }}", "{{ row.3|addslashes }}", "{{ row.4|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.

## 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.

# 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.

## Install ReportLab

Download and install the ReportLab library from http://www.reportlab.org/downloads.html. The user guide (not coincidentally, a PDF file) explains how to install it.

Test your installation by importing it in the Python interactive interpreter:

```
>>> import reportlab
```

If that command doesn't raise any errors, the installation worked.

## 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.

## Complex PDFs

If you're creating a complex PDF document with ReportLab, consider using the cStringIO library as a temporary holding place for your PDF file. The cStringIO library provides a file-like object interface that is particularly efficient. Here's the above "Hello World" example rewritten to use `cStringIO`:

```
from cStringIO import StringIO
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 = StringIO()

    # Create the PDF object, using the StringIO 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 StringIO buffer and write it to the response.
    pdf = buffer.getvalue()
    buffer.close()
    response.write(pdf)
    return response
```

## 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 HTML2PDF 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.

- forge_fdf in Python is a library that fills in PDF forms.

## 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.

# How to serve static files

Django itself doesn't serve static (media) files, such as images, style sheets, or video. It leaves that job to whichever Web server you choose.

The reasoning here is that standard Web servers, such as Apache, lighttpd and Cherokee, are much more fine-tuned at serving static files than a Web application framework.

With that said, Django does support static files **during development**. You can use the `django.views.static.serve()` view to serve media files.

## See also

If you just need to serve the admin media from a nonstandard location, see the `--adminmedia` parameter to `runserver`.

## The big, fat disclaimer

Using this method is **inefficient** and **insecure**. Do not use this in a production setting. Use this only for development.

For information on serving static files in an Apache production environment, see the *Django mod_python documentation*.

## How to do it

Here's the formal definition of the `serve()` view:

**`def serve(request, path, document_root, show_indexes=False):`**

To use it, just put this in your *URLconf*:

```
(r'^site_media/(?P<path>.*)$', 'django.views.static.serve',
        {'document_root': '/path/to/media'}),
```

...where `site_media` is the URL where your media will be rooted, and `/path/to/media` is the filesystem root for your media. This will call the `serve()` view, passing in the path from the URLconf and the (required) `document_root` parameter.

Given the above URLconf:

- The file `/path/to/media/foo.jpg` will be made available at the URL `/site_media/foo.jpg`.
- The file `/path/to/media/css/mystyles.css` will be made available at the URL `/site_media/css/mystyles.css`.
- The file `/path/bar.jpg` will not be accessible, because it doesn't fall under the document root.

Of course, it's not compulsory to use a fixed string for the `'document_root'` value. You might wish to make that an entry in your settings file and use the setting value there. That will allow you and other developers working on the code to easily change the value as required. For example, if we have a line in `settings.py` that says:

```
STATIC_DOC_ROOT = '/path/to/media'
```

...we could write the above *URLconf* entry as:

```
from django.conf import settings
...
(r'^site_media/(?P<path>.*)$', 'django.views.static.serve',
        {'document_root': settings.STATIC_DOC_ROOT}),
```

Be careful not to use the same path as your `ADMIN_MEDIA_PREFIX` (which defaults to `/media/`) as this will overwrite your URLconf entry.

## Directory listings

Optionally, you can pass the show_indexes parameter to the serve() view. This is False by default. If it's True, Django will display file listings for directories.

For example:

```
(r'^site_media/(?P<path>.*)$', 'django.views.static.serve',
        {'document_root': '/path/to/media', 'show_indexes': True}),
```

You can customize the index view by creating a template called static/directory_index.html. That template gets two objects in its context:

- directory -- the directory name (a string)
- file_list -- a list of file names (as strings) in the directory

Here's the default static/directory_index.html template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta http-equiv="Content-Language" content="en-us" />
    <meta name="robots" content="NONE,NOARCHIVE" />
    <title>Index of {{ directory }}</title>
</head>
<body>
    <h1>Index of {{ directory }}</h1>
    <ul>
    {% for f in file_list %}
    <li><a href="{{ f }}">{{ f }}</a></li>
    {% endfor %}
    </ul>
</body>
</html>
```

Changed in version 1.0.3: Prior to Django 1.0.3, there was a bug in the view that provided directory listings. The template that was loaded had to be called static/directory_listing (with no .html extension). For backwards compatibility with earlier versions, Django will still load templates with the older (no extension) name, but it will prefer a the directory_index.html version.

## Limiting use to DEBUG=True

Because URLconfs are just plain Python modules, you can use Python logic to make the static-media view available only in development mode. This is a handy trick to make sure the static-serving view doesn't slip into a production setting by mistake.

Do this by wrapping an if DEBUG statement around the django.views.static.serve() inclusion. Here's a full example URLconf:

```
from django.conf.urls.defaults import *
from django.conf import settings

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})/$', 'news.views.month_archive'),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d+)/$', 'news.views.article_detail'),
)

if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^site_media/(?P<path>.*)$', 'django.views.static.serve', {'document_root': '/path/to/media'}),
    )
```

This code is straightforward. It imports the settings and checks the value of the DEBUG setting. If it evaluates to True, then `site_media` will be associated with the `django.views.static.serve` view. If not, then the view won't be made available.

Of course, the catch here is that you'll have to remember to set `DEBUG=False` in your production settings file. But you should be doing that anyway.

### 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.

# Django FAQ

## FAQ: General

### 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.

### 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.

### Is Django stable?

Yes. World Online has been using Django for more than three years. Sites built on Django have weathered traffic spikes of over one million hits an hour and a number of Slashdottings. Yes, it's quite stable.

### 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*.

## 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*

## Which sites use Django?

The Django wiki features a consistently growing list of Django-powered sites. Feel free to add your Django-powered site to the list.

## 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.

## <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.

## 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*.

## 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!).

## 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 (ReStructured Text) 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.

## 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/ .

# FAQ: Installation

## 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.

## What are Django's prerequisites?

Django requires Python, specifically any version of Python from 2.3 through 2.6. 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, 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.

## Do I lose anything by using Python 2.3 versus newer Python versions, such as Python 2.5?

Not in the core framework. Currently, Django itself officially supports any version of Python from 2.3 through 2.6, inclusive. However, some add-on components may require a more recent Python version; the `django.contrib.gis` component, for example, requires at least Python 2.4, and third-party applications for use with Django are, of course, free to set their own version requirements.

Please note, however, that 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 3.0 (see next question for details). So if you're just starting out with Python, it's recommended that you use the latest 2.x release (currently, Python 2.6). This will let you take advantage of the numerous improvements and optimizations to the Python language since version 2.3, and will help ease the process of dropping support for older Python versions on the road to Python 3.0.

## Can I use Django with Python 3.0?

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.

## Will Django run under shared hosting (like TextDrive or Dreamhost)?

See our Django-friendly Web hosts page.

## Should I use the official version or development version?

The Django developers improve Django every day and are pretty good about not checking in broken code. We use the development code (from the Subversion repository) directly on our servers, so we consider it stable. With that in mind, we recommend that you use the latest development code, because it generally contains more features and fewer bugs than the "official" releases.

# FAQ: Using Django

## 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).
- If you're using mod_python but *not* using Django's request handler, you'll need to work around a mod_python bug related to the use of `SetEnv`; before you import anything from Django you'll need to do the following:

  ```
  os.environ.update(req.subprocess_env)
  ```

  (where `req` is the mod_python request object).

## 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.

## 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.

## 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 URL to your image in a template with `{{ object.mug_shot.url }}`.

## 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*.

# FAQ: Getting Help

## 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.

## 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.

## 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 e-mail 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.

## 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*.

## 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.

# FAQ: Databases and models

## 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 raw SQL logged in connection.queries may not include parameter quoting. Parameter quoting is performed by the database-specific backend, and not all backends provide a way to retrieve the SQL after quoting.

## Can I use Django with a pre-existing database?

Yes. See *Integrating with a legacy database*.

## 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 an option to reset the SQL for a particular application:

```
manage.py reset appname
```

This drops any tables associated with appname and recreates them.

If you do care about deleting data, you'll have to execute the ALTER TABLE statements manually in your database. That's the way we've always done it, because dealing with data is a very sensitive operation that we've wanted to avoid automating. That said, there's some work being done to add partially automated database-upgrade functionality.

## 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.

## 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.

## 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()
```

# FAQ: The admin

## 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.

## 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.

## 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.

## 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 `ModelAdmin.save_model()` hook, you can update an object to reflect the user that edited it. See *the documentation on ModelAdmin methods* for an example.

## 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 `ModelAdmin.queryset()` and `ModelAdmin.has_change_permission()` can be used to control the visibility and editability of objects in the admin.

## My admin-site CSS and images showed up fine using the development server, but they're not displaying when using mod_python.

See *serving the admin files* in the "How to use Django with mod_python" documentation.

## 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.

## 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.

## 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.

# FAQ: Contributing code

## 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*.

## 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.

## *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.

## *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.

# API Reference

# *Built-in authentication backends reference*

This document details the authentication backends that come with Django. For information on how how to use them and how to write your own authentication backends, see the *Other authentication sources section* of the *User authentication guide*.

## *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`**

New in version 1.1: *Please, see the release notes*

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.

# The "django.contrib" add-ons

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`.

## 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.

> **Note**
>
> The admin site has been refactored significantly since Django 0.96. This document describes the newest version of the admin site, which allows for much richer customization. If you follow the development of Django itself, you may have heard this described as "newforms-admin."

### Overview

There are five steps in activating the Django admin site:

1. Add `django.contrib.admin` to your `INSTALLED_APPS` setting.
2. Determine which of your application's models should be editable in the admin interface.
3. For each of those models, optionally create a `ModelAdmin` class that encapsulates the customized admin functionality and options for that particular model.
4. Instantiate an `AdminSite` and tell it about each of your models and `ModelAdmin` classes.
5. Hook the `AdminSite` instance into your URLconf.

### Other topics

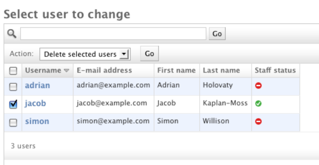
### Admin actions

New in version 1.1: *Please, see the release notes*

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 explicitally 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':
            del actions['delete_selected']
        return actions
```

### See also

For information about serving the media files (images, JavaScript, and CSS) associated with the admin in production, see *Serving media 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.**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'
```

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.

ModelAdmin.**fieldsets**

Set fieldsets to control the layout of admin "add" and "change" pages.

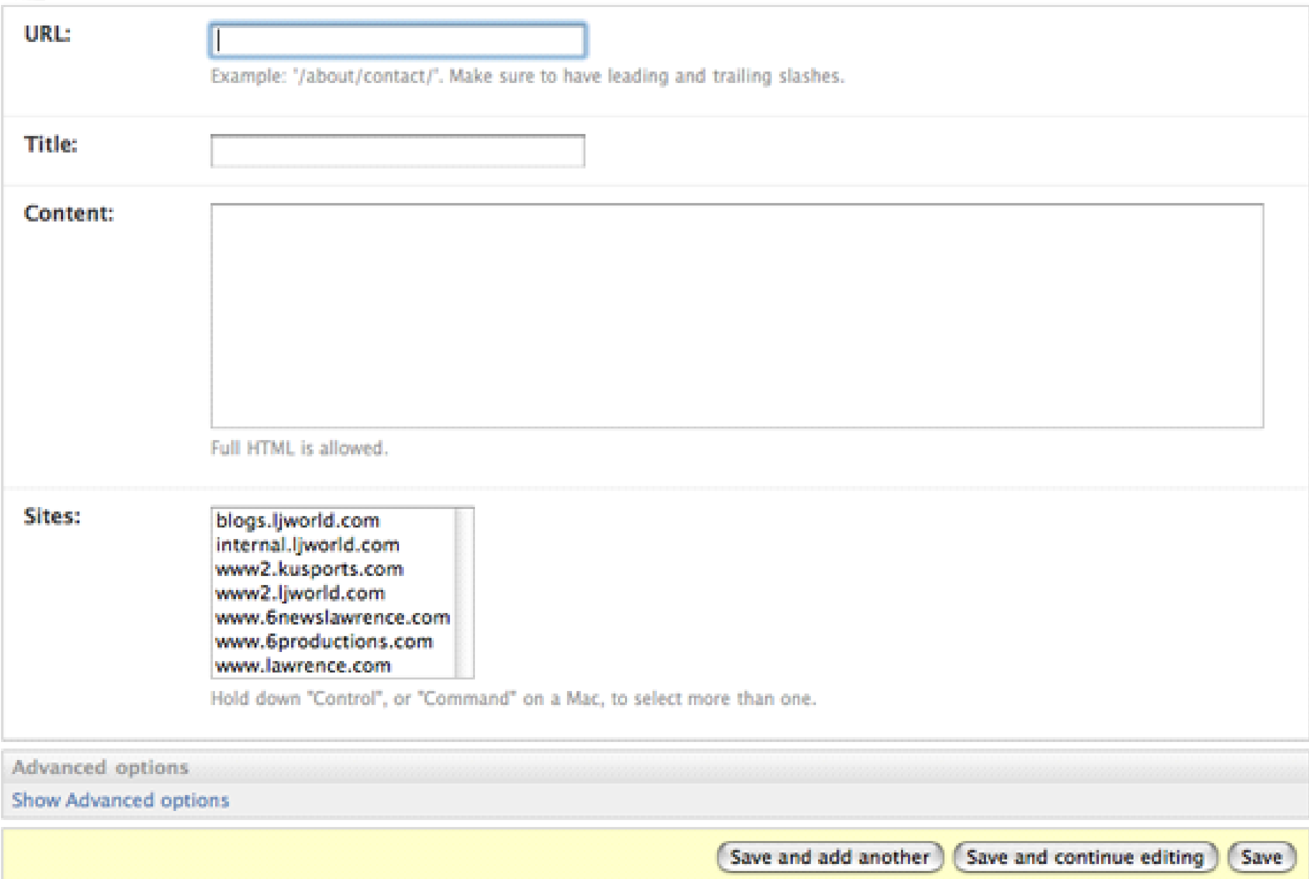
fieldsets is a list of two-tuples, in which each two-tuple represents a <fieldset> on the admin form page. (A <fieldset> is a "section" of the form.)

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 fieldsets isn't given, 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'),
  }
  ```

  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'),
  }
  ```

- **`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.`**`fields`**

Use this option as an alternative to `fieldsets` if the layout does not matter and if you want to only show a subset of the available fields in the form. 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.

> ## Note
>
> This `fields` option should not be confused with the `fields` dictionary key that is within the `fieldsets` option, as described in the previous section.

`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.`**`filter_horizontal`**

Use a nifty unobtrusive JavaScript "filter" interface instead of the usability-challenged `<select multiple>` in the admin form. The value is a list of fields that should be displayed as a horizontal filter interface. See `filter_vertical` to use a vertical interface.

`ModelAdmin.`**`filter_vertical`**

Same as `filter_horizontal`, but is a vertical display of the filter interface.

`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, self.last_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 field names (in the same format as `list_display`) to link.

`list_display_links` can specify one or many field names. As long as the field names 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`**

New in version 1.1: *Please, see the release notes*

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.

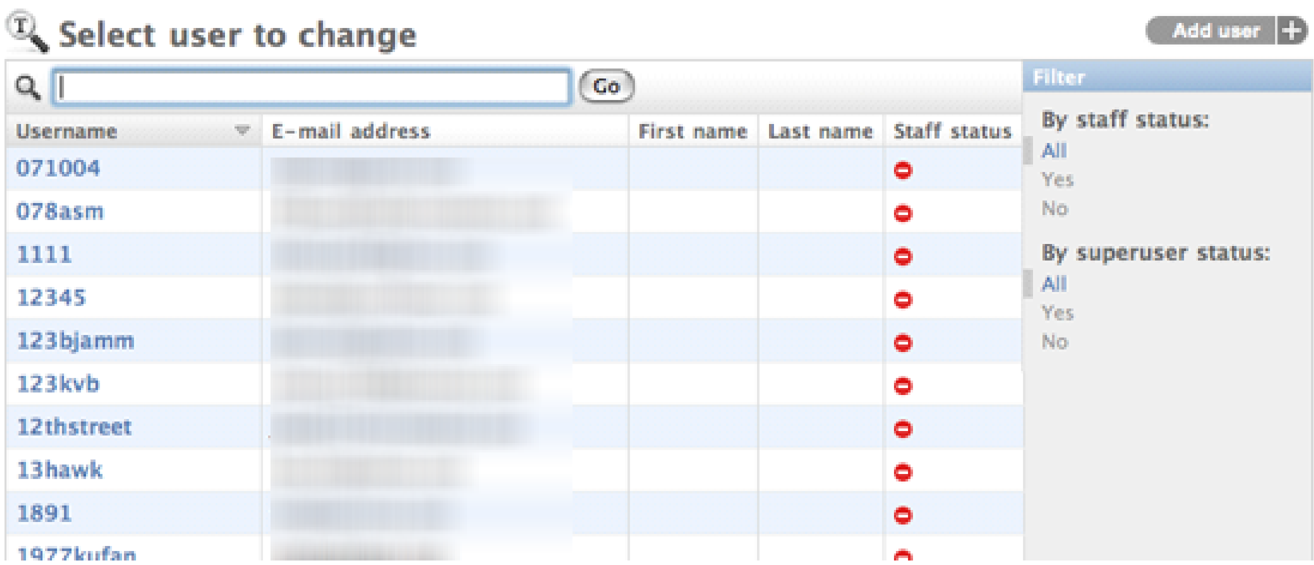
`ModelAdmin.`**`list_filter`**

Set `list_filter` to activate filters in the right sidebar of the change list page of the admin. This should be a list of field names, and each specified field should be either a `BooleanField`, `CharField`, `DateField`, `DateTimeField`, `IntegerField` or `ForeignKey`.

This example, taken from the `django.contrib.auth.models.User` model, shows how both `list_display` and `list_filter` work:

```
class UserAdmin(admin.ModelAdmin):
    list_display = ('username', 'email', 'first_name', 'last_name', 'is_staff')
    list_filter = ('is_staff', 'is_superuser')
```

The above code results in an admin change list page that looks like this:



(This example also has search_fields defined. See below.)

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.

For more on select_related(), see *the select_related() docs*.

ModelAdmin.**inlines**

See InlineModelAdmin objects below.

ModelAdmin.**ordering**

Set ordering to specify how objects on the admin change list page should be ordered. 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.

> ***Note***
>
> Django will only honor the first element in the list/tuple; any others will be ignored.

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 a Input widget for either a ForeignKey or ManyToManyField:

```
class ArticleAdmin(admin.ModelAdmin):
    raw_id_fields = ("newspaper",)
```

`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` with the lookup API "follow" notation:

```
search_fields = ['foreign_key__related_fieldname']
```

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.

`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.`**`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.`**`change_list_template`**

Path to a custom template that will be used by the model objects "change list" view. Templates can override or extend base admin templates as described in Overriding Admin Templates.

If you don't specify this attribute, a default template shipped with Django that provides the standard appearance is used.

`ModelAdmin.`**`change_form_template`**

Path to a custom template that will be used by both the model object creation and change views. Templates can override or extend base admin templates as described in Overriding Admin Templates.

If you don't specify this attribute, a default template shipped with Django that provides the standard appearance is used.

`ModelAdmin.`**`object_history_template`**

Path to a custom template that will be used by the model object change history display view. Templates can override or extend base admin templates as described in Overriding Admin Templates.

If you don't specify this attribute, a default template shipped with Django that provides the standard appearance is used.

`ModelAdmin.`**`delete_confirmation_template`**

Path to a custom template that will be used by the view responsible of showing the confirmation page when the user decides to delete one or more model objects. Templates can override or extend base admin templates as described in Overriding Admin Templates.

If you don't specify this attribute, a default template shipped with Django that provides the standard appearance is used.

## *ModelAdmin* methods

**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.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_urls (*self*)**

New in version 1.1: *Please, see the release notes*

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
```

### 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.

However, the self.my_view function registered above suffers from two problems:

- It will *not* perform and 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*)**

New in version 1.1: *Please, see the release notes*

The `formfield_for_foreignkey` method on a `ModelAdmin` allows you to override the default formfield for a foreign key 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 db_field.formfield(**kwargs)
        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 the cars owned by the `User` instance.

## 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*, *extra_context=None*)**

Django view for the model instance edition page. See note below.

**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 comon 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):
        # ...

    def change_view(self, request, object_id, extra_context=None):
        my_context = {
            'osm_data': self.get_osm_info(),
        }
        return super(MyModelAdmin, self).change_view(request, object_id,
            extra_context=my_context)
```

## `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",)
```

Keep in mind that this will be prepended with `MEDIA_URL`. The same rules apply as *regular media definitions on forms*.

## 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

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

The InlineModelAdmin class is a subclass of ModelAdmin so it inherits all the same functionality as well as some of its own:

### *model*

The model in which the inline is using. This is required.

### *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.

### *formset*

This defaults to BaseInlineFormSet. Using your own formset can give you many possibilities of customization. Inlines are built around *model formsets*.

### *form*

The value for form defaults to BaseModelForm. This is what is passed through to formset_factory when creating the formset for this inline.

### *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.

### *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.

### *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",)
```

### *template*

The template used to render the inline on the page.

### *verbose_name*

An override to the `verbose_name` found in the model's inner `Meta` class.

### *verbose_name_plural*

An override to the `verbose_name_plural` found in the model's inner `Meta` class.

## *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 Intermediary Models*

By default, admin widgets for many-to-many relations will be displayed inline on whichever model contains the actual reference to the `ManyToManyField`. However, 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 simply use `GenericInlineModelAdmin` provided by `django.contrib.contenttypes.generic`. 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)
```

`django.contrib.contenttypes.generic` provides both a `GenericTabularInline` and `GenericStackedInline` and behave just like any other inline. 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 or login templates, you are better off creating your own AdminSite instance (see below), and changing the AdminSite.index_template or AdminSite.login_template properties.

## AdminSite objects

*class* **AdminSite** (*name=None*)

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.

New in version 1.1: *Please, see the release notes*

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

AdminSite.**index_template**

Path to a custom template that will be used by the admin site main index view. Templates can override or extend base admin templates as described in Overriding Admin Templates.

AdminSite.**login_template**

Path to a custom template that will be used by the admin site login view. Templates can override or extend base admin templates as described in Overriding Admin Templates.

## 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.defaults import *
from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',
    ('^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.defaults import *
from myproject.admin import admin_site

urlpatterns = patterns('',
    ('^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.

Changed in version 1.1: The method for hooking AdminSite instances into urls has changed in Django 1.1.

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.defaults import *
from myproject.admin import basic_site, advanced_site

urlpatterns = patterns('',
    ('^basic-admin/', include(basic_site.urls)),
    ('^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

New in version 1.1: *Please, see the release notes*

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.

## Reversing Admin URLs

New in version 1.1: *Please, see the release notes*

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 |

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*.

# django.contrib.auth

See *User authentication in Django*.

# 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 get a 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.

## 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 %}
<form action="{% comment_form_target %}" method="POST">
  {{ form }}
  <tr>
    <td></td>
    <td><input type="submit" name="preview" class="submit-post" value="Preview"></td>
  </tr>
</form>
```

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.

# 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 posteed 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
```

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 valdation 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. 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. 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.

## 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).

# 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* models.**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

*class* models.**ContentType**

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:

**models.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.

**models.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* `models.`**`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 `django.contrib.contenttypes.models.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*)**

Takes either a model class or an instance of a model, and returns the `ContentType` instance representing that model.

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>
```

## 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 -- `django.contrib.contenttypes.generic.GenericForeignKey` -- which works around this and allows the relationship to be with any model. 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 a primary-key value from the models you'll be relating to. (For most models, this means an `IntegerField` or `PositiveIntegerField`.)

   This field must be of the same type as the primary key of the models that will be involved in the generic relation. For example, if you use `IntegerField`, you won't be able to form a generic relation with a model that uses a `CharField` as a primary key.
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.

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. They aren't normal field objects. 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

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='http://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 django.contrib.contenttypes.generic.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='http://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 with a GenericForeignKey that you're referring to uses a non-default value for ct_field or fk_field (e.g. the django.contrib.comments app uses ct_field="object_pk"), you'll need to pass content_type_field and object_id_field to GenericRelation.:

```
comments = generic.GenericRelation(Comment, content_type_field="content_type", object_id_field="object_pk")
```

Note 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.

### 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

django.contrib.contenttypes.generic provides both a GenericInlineFormSet and GenericInlineModelAdmin. This enables the use of generic relations in forms and the admin. See the *model formset* and *admin* documentation for more information.

*class* generic.**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.

## Cross Site Request Forgery protection

The CsrfMiddleware class provides easy-to-use protection against Cross Site Request Forgeries. This type of attack occurs when a malicious Web site creates a link or form button that is intended to perform some action on your Web site, using the credentials of a logged-in user who is tricked into clicking on the link in their browser.

The first defense against CSRF attacks is to ensure that GET requests are side-effect free. POST requests can then be protected by adding this middleware into your list of installed middleware.

## How to use it

Add the middleware `'django.contrib.csrf.middleware.CsrfMiddleware'` to your list of middleware classes, `MIDDLEWARE_CLASSES`. It needs to process the response after the SessionMiddleware, so must come before it in the list. It also must process the response before things like compression happen to the response, so it must come after GZipMiddleware in the list.

The `CsrfMiddleware` class is actually composed of two middleware: `CsrfViewMiddleware` which performs the checks on incoming requests, and `CsrfResponseMiddleware` which performs post-processing of the result. This allows the individual components to be used and/or replaced instead of using `CsrfMiddleware`.

Changed in version 1.1: (previous versions of Django did not provide these two components of `CsrfMiddleware` as described above)

## Exceptions

New in version 1.1: *Please, see the release notes*

To manually exclude a view function from being handled by the CsrfMiddleware, you can use the `csrf_exempt` decorator, found in the `django.contrib.csrf.middleware` module. For example:

```
from django.contrib.csrf.middleware import csrf_exempt

def my_view(request):
    return HttpResponse('Hello world')
my_view = csrf_exempt(my_view)
```

Like the middleware itself, the `csrf_exempt` decorator is composed of two parts: a `csrf_view_exempt` decorator and a `csrf_response_exempt` decorator, found in the same module. These disable the view protection mechanism (`CsrfViewMiddleware`) and the response post-processing (`CsrfResponseMiddleware`) respectively. They can be used individually if required.

You don't have to worry about doing this for most AJAX views. Any request sent with "X-Requested-With: XMLHttpRequest" is automatically exempt. (See the next section.)

## How it works

CsrfMiddleware does two things:

1. It modifies outgoing requests by adding a hidden form field to all 'POST' forms, with the name 'csrfmiddlewaretoken' and a value which is a hash of the session ID plus a secret. If there is no session ID set, this modification of the response isn't done, so there is very little performance penalty for those requests that don't have a session. (This is done by `CsrfResponseMiddleware`).
2. On all incoming POST requests that have the session cookie set, it checks that the 'csrfmiddlewaretoken' is present and correct. If it isn't, the user will get a 403 error. (This is done by `CsrfViewMiddleware`)

This ensures that only forms that have originated from your Web site can be used to POST data back.

It deliberately only targets HTTP POST requests (and the corresponding POST forms). GET requests ought never to have any potentially dangerous side effects (see 9.1.1 Safe Methods, HTTP 1.1, RFC 2616), and so a CSRF attack with a GET request ought to be harmless.

POST requests that are not accompanied by a session cookie are not protected, but they do not need to be protected, since the 'attacking' Web site could make these kind of requests anyway.

The Content-Type is checked before modifying the response, and only pages that are served as 'text/html' or 'application/xml+x-html' are modified.

The middleware tries to be smart about requests that come in via AJAX. Many JavaScript toolkits send an "X-Requested-With: XMLHttpRequest" HTTP header; these requests are detected and automatically *not* handled by this middleware. We can do this safely because, in the context of a browser, the header can only be added by using `XMLHttpRequest`, and browsers already implement a same-domain policy for `XMLHttpRequest`. (Note that this is not secure if you don't trust content within the same domain or subdomains.)

## Limitations

CsrfMiddleware requires Django's session framework to work. If you have a custom authentication system that manually sets cookies and the like, it won't help you.

If your app creates HTML pages and forms in some unusual way, (e.g. it sends fragments of HTML in JavaScript document.write statements) you might bypass the filter that adds the hidden field to the form, in which case form submission will always fail. It may still be possible to use the middleware, provided you can find some way to get the CSRF token and ensure that is included when your form is submitted.

# Databrowse

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.

> ## Note
>
> Databrowse is **very** new and is currently under active development. It may change substantially before the next Django release.
>
> With that said, it's easy to use, and it doesn't require writing any code. So you can play around with it today, with very little investment in time or coding.

## 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

   databrowse.site.register(SomeModel)
   databrowse.site.register(SomeOtherModel)
   ```

   Note that you should register the model *classes*, not instances.

   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.

# 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.

New in version 1.0: *Please, see the release notes*

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.chicagocrime.org/about/
- http://www.everyblock.com/about/
- http://www.lawrence.com/about/contact/

## 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 not 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. 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.

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 `FlatpageFallbackMiddleware` at the end of the list, because it's a last resort.

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* `models.`**`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*.

## Flatpage templates

By default, flatpages are rendered via the template `flatpages/default.html`, but you can override that for a particular flatpage.

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 PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">
<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.

## 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.

# Form wizard

New in version 1.0: *Please, see the release notes*

Django comes with an optional "form wizard" application that splits *forms* across multiple Web pages. It maintains state in hashed HTML `<input type="hidden">` fields, and the data isn't processed server-side until the final form is submitted.

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 calculates a secure hash of the data and presents the user with the next form, saving the validated data and hash in `<input type="hidden">` fields.
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 e-mail, 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 `django.forms` `Form` classes -- one per wizard page.
2. Create a `FormWizard` class 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. Point your URLconf at your `FormWizard` class.

## 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 e-mail 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)
```

**Important limitation:** Because the wizard uses HTML hidden fields to store data between pages, you may not include a `FileField` in any form except the last one.

## Creating a `FormWizard` class

The next step is to create a `FormWizard` class, which should be a subclass of `django.contrib.formtools.wizard.FormWizard`.

As with your `Form` classes, this `FormWizard` class can live anywhere in your codebase, but convention is to put it in `forms.py`.

The only requirement on this subclass is that it implement a `done()` method, which specifies what should happen when the data for *every* form is submitted and validated. This method is passed two arguments:

- `request` -- an `HttpRequest` object
- `form_list` -- a list of `django.forms` `Form` classes

In this simplistic example, rather than perform 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 import FormWizard

class ContactWizard(FormWizard):
    def done(self, request, form_list):
        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 import FormWizard

class ContactWizard(FormWizard):
    def done(self, request, form_list):
        do_something_with_the_form_data(form_list)
        return HttpResponseRedirect('/page-to-redirect-to-when-done/')
```

See the section Advanced FormWizard methods below to learn about more `FormWizard` 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 `forms/wizard.html`. (You can change this template name by overriding `get_template()`, which is documented below. This hook also allows you to use a different template for each form.)

This template expects the following context:

- `step_field` -- The name of the hidden field containing the step.
- `step0` -- The current step (zero-based).
- `step` -- The current step (one-based).
- `step_count` -- The total number of steps.
- `form` -- The `Form` instance for the current step (either empty or with errors).
- `previous_fields` -- A string representing every previous data field, plus hashes for completed forms, all in the form of hidden fields. Note that you'll need to run this through the `safe()` template filter, to prevent auto-escaping, because it's raw HTML.

It will also be passed any objects in `extra_context`, which is a dictionary you can specify that contains extra values to add to the context. You can specify it in two ways:

- Set the `extra_context` attribute on your `FormWizard` subclass to a dictionary.
- Pass `extra_context` as extra parameters in the URLconf.

Here's a full example template:

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

{% block content %}
<p>Step {{ step }} of {{ step_count }}</p>
<form action="." method="post">
<table>
{{ form }}
</table>
<input type="hidden" name="{{ step_field }}" value="{{ step0 }}" />
{{ previous_fields|safe }}
<input type="submit">
</form>
{% endblock %}
```

Note that `previous_fields`, `step_field` and `step0` are all required for the wizard to work properly.

## Hooking the wizard into a URLconf

Finally, give your new FormWizard object a URL in `urls.py`. The wizard takes a list of your form objects as arguments:

```
from django.conf.urls.defaults import *
from mysite.testapp.forms import ContactForm1, ContactForm2, ContactWizard

urlpatterns = patterns('',
    (r'^contact/$', ContactWizard([ContactForm1, ContactForm2])),
)
```

## Advanced FormWizard methods

*class* **FormWizard**

Aside from the `done()` method, FormWizard 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 representing the current step of the wizard. (E.g., the first form is `0` and the second form is `1`.)

**FormWizard.prefix_for_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*.

Default implementation:

```
def prefix_for_step(self, step):
    return str(step)
```

**FormWizard.render_hash_failure ()**

Renders a template if the hash check fails. It's rare that you'd need to override this.

Default implementation:

```
def render_hash_failure(self, request, step):
    return self.render(self.get_form(step), request, step,
        context={'wizard_error': 'We apologize, but your form has expired. Please continue filling out the form from this page.'})
```

**FormWizard.security_hash ()**

Calculates the security hash for the given request object and `Form` instance.

By default, this uses an MD5 hash of the form data and your SECRET_KEY setting. It's rare that somebody would need to override this.

Example:

```
def security_hash(self, request, form):
    return my_hash_function(request, form)
```

**FormWizard.parse_params ()**

A hook for saving state from the request object and `args` / `kwargs` that were captured from the URL by your URLconf.

By default, this does nothing.

Example:

```
def parse_params(self, request, *args, **kwargs):
    self.my_state = args[0]
```

**FormWizard.get_template ()**

Returns the name of the template that should be used for the given step.

By default, this returns `'forms/wizard.html'`, regardless of step.

Example:

```
def get_template(self, step):
    return 'myapp/wizard_%s.html' % step
```

If `get_template()` returns a list of strings, then the wizard will use the template system's `select_template()` function, *explained in the template docs*. This means the system will use the first template that exists on the filesystem. For example:

```
def get_template(self, step):
    return ['myapp/wizard_%s.html' % step, 'myapp/wizard.html']
```

**FormWizard.render_template ()**

Renders the template for the given step, returning an `HttpResponse` object.

Override this method if you want to add a custom context, return a different MIME type, etc. If you only need to override the template name, use `get_template()` instead.

The template will be rendered with the context documented in the "Creating templates for the forms" section above.

**FormWizard.process_step ()**

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 should *not* modify any of that data. Rather, it might want to set `self.extra_context` or dynamically alter `self.form_list`, based on previously submitted forms.

Note that this method is called every time a page is rendered for *all* submitted steps.

The function signature:

```
def process_step(self, request, form, step):
    # ...
```

# 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 these 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'`.

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 1000000000000000 (one quadrillion) are supported.

You can pass in either an integer or a string representation of an integer.

## 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.

## naturalday

New in version 1.0: *Please, see the release notes*

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 now 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.

# 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
- Brazil
- Canada
- Chile
- Czech
- Finland
- France
- Germany

- Iceland
- India
- Italy
- Japan
- Mexico
- The Netherlands
- Norway
- Peru
- Poland
- Romania
- Slovakia
- South Africa
- Spain
- Switzerland
- United Kingdom
- United States of America

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()
```

## 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.

## 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)

*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.

## Austria (at)

*class* `at.forms.`**`ATZipCodeField`**

A form field that validates its input as an Austrian zip code.

*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.

## 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.

## 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.

## 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.

## *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.

## *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.

## 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.

## Mexico (mx)

*class* `mx.forms.`**`MXStateSelect`**

A `Select` widget that uses a list of Mexican states as its choices.

## 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.

## Peru (pe)

*class* `pt.forms.`**`PEDNIField`**

A form field that validates input as a DNI (Peruvian national identity) number.

*class* `pt.forms.`**`PERUCField`**

A form field that validates input as an RUC (Registro Unico de Contribuyentes) number. Valid RUC numbers have 11 digits.

*class* `pt.forms.`**`PEDepartmentSelect`**

A `Select` widget that uses a list of Peruvian Departments as its choices.

## Poland (pl)

*class* `pl.forms.`**`PLNationalIdentificationNumberField`**

A form field that validates input as a Polish national identification number (PESEL).

*class* `pl.forms.`**`PLNationalBusinessRegisterField`**

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.`**`PLTaxNumberField`**

A form field that validates input as a Polish Tax Number (NIP). Valid formats are XXX-XXX-XX-XX or XX-XX-XXX-XXX. The checksum algorithm used for NIPs is documented at http://wipos.p.lodz.pl/zylla/ut/nip-rego.html.

*class* `pl.forms.`**`PLAdministrativeUnitSelect`**

A `Select` widget that uses a list of Polish administrative units as its choices.

*class* `pl.forms.`**`PLVoivodeshipSelect`**

A `Select` widget that uses a list of Polish voivodeships (administrative provinces) as its choices.

## 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.

## 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.

## 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.

## 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 -- see http://adi.kousz.ch/artikel/IDCHE.htm.

*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.

## United Kingdom (uk)

*class* `uk.forms.`**`UKPostcodeField`**

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.govtalk.gov.uk/gdsc/schemas/bs7666-v2-0.xsd.

*class* `uk.forms.`**`UKCountySelect`**

A `Select` widget that uses a list of UK counties/regions as its choices.

*class* `uk.forms.`**`UKNationSelect`**

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.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.

## 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*.

# 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.load_template_source' 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 load_template_source() 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 mysite.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 URL for a given object as returned by `items()`.

If it's an attribute, its value should be a string representing an absolute URL to use for *every* object returned by `items()`.

In both cases, "absolute URL" 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()`.

**`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.

## Shortcuts

The sitemap framework provides a couple convenience classes for common cases:

*class* **`FlatPageSitemap`**

The `django.contrib.sitemaps.FlatPageSitemap` class looks at all `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 works with any *generic views* you already have. To use it, create an instance, passing in the same `info_dict` you pass to the generic views. The only requirement is that the dictionary have a `queryset` entry. 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.defaults import *
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from mysite.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:

```
(r'^sitemap.xml$', 'django.contrib.sitemaps.views.index', {'sitemaps': sitemaps}),
(r'^sitemap-(?P<section>.+)\.xml$', 'django.contrib.sitemaps.views.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.

## 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 URL of 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*

New in version 1.0: *Please, see the release notes*

Once the sitemaps application is added to your project, you may also ping the Google server's through the command line manage.py interface:

```
python manage.py ping_google [/sitemap.xml]
```

## 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* `django.contrib.sites.models.`**`Site`**

This model has `domain` and `name` fields. The `SITE_ID` setting specifies the database ID of the `Site` object associated with that particular settings file.

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*

On a lower level, 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.
```

## *Getting the current domain for display*

LJWorld.com and Lawrence.com both have e-mail 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 e-mail 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 e-mail has the subject line "Thanks for subscribing to lawrence.com alerts." On LJWorld.com, the e-mail has the subject "Thanks for subscribing to LJWorld.com alerts." Same goes for the e-mail'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

New in version 1.0: *Please, see the release notes*

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* `django.contrib.sites.managers.`**`CurrentSiteManager`**

If `Site`s play 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`? It defaults to looking for a field called `Site`. If your model has a `ForeignKey` or `ManyToManyField` called something *other* than `Site`, you need to explicitly pass that as the parameter to `CurrentSiteManager`. 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

New in version 1.0: *Please, see the release notes*

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.

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`.

# 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 a view that's hooked to `/feeds/` by default. Django uses the remainder of the URL (everything after `/feeds/`) to determine which feed to output.

To create a feed, just write a Feed class and point to it in your *URLconf*.

### Initialization

To activate syndication feeds on your Django site, add this line to your *URLconf*:

```
(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed', {'feed_dict': feeds}),
```

This tells Django to use the RSS framework to handle all URLs starting with `"feeds/"`. (You can change that `"feeds/"` prefix to fit your own needs.)

This URLconf line has an extra argument: `{'feed_dict': feeds}`. Use this extra argument to pass the syndication framework the feeds that should be published under that URL.

Specifically, `feed_dict` should be a dictionary that maps a feed's slug (short URL label) to its Feed class.

You can define the `feed_dict` in the URLconf itself. Here's a full example 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}),
    # ...
)
```

The above example registers two feeds:

- The feed represented by `LatestEntries` will live at `feeds/latest/`.
- The feed represented by `LatestEntriesByCategory` will live at `feeds/categories/`.

Once that's set up, you just need to define the Feed classes themselves.

## Feed classes

A Feed class is a simple 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 must subclass `django.contrib.syndication.feeds.Feed`. They can live anywhere in your codebase.

## A simple example

This simple example, taken from chicagocrime.org, describes a feed of the latest five news items:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem

class LatestEntries(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]
```

Note:

- The class subclasses `django.contrib.syndication.feeds.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's 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.

- To specify the contents of `<title>` and `<description>`, create *Django templates* called `feeds/latest_title.html` and `feeds/latest_description.html`, where `latest` is the `slug` specified in the URLconf for the given feed. Note the `.html` extension is required. The RSS system renders that template for each item, passing it 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.

  If you don't create a template for either the title or description, the framework will use the template `"{{ obj }}"` by default -- that is, the normal string representation of the object. You can also change the names of these two templates by specifying `title_template` and `description_template` as attributes of your Feed class.
- To specify the contents of `<link>`, you have two options. For each item in `items()`, Django first tries calling a method `item_link()` in the Feed class, passing it a single parameter, `item`, which is the object itself. 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.
- For the LatestEntries example above, we could have very simple feed templates:
    - latest_title.html:

      ```
      {{ obj.title }}
      ```
    - latest_description.html:

      ```
      {{ obj.description }}
      ```

## *A complex example*

The framework also supports more complex feeds, via parameters.

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 make generic feeds that output items based on information in the feed's URL.

On chicagocrime.org, the police-beat feeds are accessible via URLs like this:

- `/rss/beats/0613/` -- Returns recent crimes for beat 0613.
- `/rss/beats/1424/` -- Returns recent crimes for beat 1424.

The slug here is `"beats"`. The syndication framework sees the extra URL bits after the slug -- `0613` and `1424` -- and gives you a hook to tell it what those URL bits mean, and how they should influence which items get published in the feed.

An example makes this clear. Here's the code for these beat-specific feeds:

```
from django.contrib.syndication.feeds import FeedDoesNotExist
from django.core.exceptions import ObjectDoesNotExist

class BeatFeed(Feed):
    def get_object(self, bits):
        # In case of "/rss/beats/0613/foo/bar/baz/", or other such clutter,
        # check that bits has only one member.
        if len(bits) != 1:
            raise ObjectDoesNotExist
        return Beat.objects.get(beat__exact=bits[0])

    def title(self, obj):
        return "Chicagocrime.org: Crimes for beat %s" % obj.beat

    def link(self, obj):
        if not obj:
            raise FeedDoesNotExist
        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__id__exact=obj.id).order_by('-crime_date')[:30]
```

Here's the basic algorithm the RSS framework follows, given this class and a request to the URL `/rss/beats/0613/`:

- The framework gets the URL `/rss/beats/0613/` and notices there's an extra bit of URL after the slug. It splits that remaining string by the slash character (`"/"`) and calls the Feed class' `get_object()` method, passing it the bits. In this case, bits is `['0613']`. For a request to `/rss/beats/0613/foo/bar/`, bits would be `['0613', 'foo', 'bar']`.
- `get_object()` is responsible for retrieving the given beat, from the given `bits`. In this case, it uses the Django database API to retrieve the beat. Note that `get_object()` should raise `django.core.exceptions.ObjectDoesNotExist` if given invalid parameters. There's no `try`/`except` around the `Beat.objects.get()` call, because it's not necessary; that function raises `Beat.DoesNotExist` on failure, and `Beat.DoesNotExist` is a subclass of `ObjectDoesNotExist`. Raising `ObjectDoesNotExist` in `get_object()` tells Django to produce a 404 error for that request.

  New in version 1.0: `get_object()` can handle the `/rss/beats/` url.

  The `get_object()` method also has a chance to handle the `/rss/beats/` url. In this case, `bits` will be an empty list. In our example, `len(bits) != 1` and an `ObjectDoesNotExist` exception will be raised, so `/rss/beats/` will generate a 404 page. But you can handle this case however you like. For example, you could generate a combined feed for all beats.

- 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.

  Inside the `link()` method, we handle the possibility that `obj` might be `None`, which can occur when the URL isn't fully specified. In some cases, you might want to do something else in this case, which would mean you'd need to check for `obj` existing in other methods as well. (The `link()` method is called very early in the feed generation process, so it's a good place to bail out early.)
- Finally, note that `items()` in this example also takes the `obj` argument. The algorithm for `items` is the same as described in the previous step -- first, it tries `items(obj)()`, then `items()`, then finally an `items` class attribute (which should be a list).

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 URL (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.feeds 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.defaults import *
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

feeds = {
    'rss': RssSiteNewsFeed,
    'atom': AtomSiteNewsFeed,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
        {'feed_dict': feeds}),
    # ...
)
```

## *Feed class reference*

*class* `django.contrib.syndication.feeds.`**Feed**

This example illustrates all possible attributes and methods for a Feed class:

```
from django.contrib.syndication.feeds 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 you don't specify one, or either, Django will use the template
    # 'feeds/SLUG_title.html' and 'feeds/SLUG_description.html', where SLUG
    # is the slug you specify in the URL.

    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 feed's
        link as a normal Python string.
        """

    def link(self):
        """
        Returns the feed's link as a normal Python string.
        """

    link = '/foo/bar/' # Hard-coded link.

    # 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 e-mail as a normal Python string.
        """

    def author_email(self):
        """
        Returns the feed's author's e-mail as a normal Python string.
        """

    author_email = 'test@example.com' # Hard-coded author e-mail.

    # 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 copyright(self, obj):
        """
        Takes the object returned by get_object() and returns the feed's
        copyright notice as a normal Python string.
        """

    def copyright(self):
        """
        Returns the feed's copyright notice as a normal Python string.
        """

    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, bits):
        """
        Takes a list of strings gleaned from the URL and returns an object
        represented by this feed. Raises
        django.core.exceptions.ObjectDoesNotExist on error.
        """

    # 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 e-mail as a normal Python string.
        """

    def item_author_email(self):
        """
        Returns the author e-mail for every item in the feed.
        """

    item_author_email = 'test@example.com' # Hard-coded author e-mail.

    # 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:

*class* `django.utils.feedgenerator.`**`SyndicationFeed`**

and several subclasses:

*class* `django.utils.feedgenerator.`**`RssUserland091Feed`**

*class* `django.utils.feedgenerator.`**`Rss201rev2Feed`**

*class* `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__ (**`*`kwargs`***`)`**

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 (**kwargs)`**

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 `feedgenerator.Enclosure`.
- `categories` should be a sequence of Unicode objects.

**`SyndicationFeed.write (outfile, encoding)`**

Outputs the feed in the given encoding to outfile, which is a file-like object.

**`SyndicationFeed.writeString (encoding)`**

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
>>> 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")
>>> f.add_item(title=u"Hot dog today",
...     link=u"http://www.example.com/entries/1/",
...     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.

## 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.

## *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.

## *auth*

Django's authentication framework.

See *User authentication in Django*.

## *comments*

Changed in version 1.0: The comments application has been rewriten. See *Upgrading from Django's previous comment system* for information on howto upgrade.

A simple yet flexible comments system. See *Django's comments framework*.

## *contenttypes*

A light framework for hooking into "types" of content, where each installed Django model is a separate content type.

See the *contenttypes documentation*.

## *csrf*

A middleware for preventing Cross Site Request Forgeries

See the *csrf documentation*.

## *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.

## *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*.

## *humanize*

A set of Django template filters useful for adding a "human touch" to data.

See the *humanize documentation*.

## *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*.

## *markup*

A collection of template filters that implement common markup languages:

- `textile` -- implements Textile
- `markdown` -- implements Markdown
- `restructuredtext` -- implements ReST (ReStructured Text)

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.

## *redirects*

A framework for managing redirects.

See the *redirects documentation*.

## *sessions*

A framework for storing data in anonymous sessions.

See the *sessions documentation*.

## *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*.

## *sitemaps*

A framework for generating Google sitemap XML files.

See the *sitemaps documentation*.

## *syndication*

A framework for generating syndication feeds, in RSS and Atom, quite easily.

See the *syndication documentation*.

## webdesign

Helpers and utilities targeted primarily at Web *designers* rather than Web *developers*.

See the *Web design helpers documentation*.

## 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.

# Notes about supported 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.

## PostgreSQL notes

### 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.

### 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

New in version 1.1: *Please, see the release notes*

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, if you're using the `postgresql_psycopg2` backend, 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 `DATABASE_OPTIONS` setting:

```
DATABASE_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 *django.db.transaction.autocommit* decorator. That decorator is a Django-level implementation that commits automatically after data changing operations. The feature enabled using the `DATABASE_OPTIONS` settings 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.

## MySQL notes

Django expects the database to support transactions, referential integrity, and Unicode (UTF-8 encoding). Fortunately, MySQL has all these features as available as far back as 3.23. While it may be possible to use 3.23 or 4.0, you'll probably have less trouble if you use 4.1 or 5.0.

### MySQL 4.1

MySQL 4.1 has greatly improved support for character sets. It is possible to set different default character sets on the database, table, and column. Previous versions have only a server-wide character set setting. It's also the first version where the character set can be changed on the fly. 4.1 also has support for views, but Django currently doesn't use views.

### MySQL 5.0

MySQL 5.0 adds the `information_schema` database, which contains detailed data on all database schema. Django's `inspectdb` feature uses this `information_schema` if it's available. 5.0 also has support for stored procedures, but Django currently doesn't use stored procedures.

### Storage engines

MySQL has several storage engines (previously called table types). You can change the default storage engine in the server configuration.

The default engine is MyISAM [1]. The main drawback of MyISAM is that it doesn't currently support transactions or foreign keys. On the plus side, it's currently the only engine that supports full-text indexing and searching.

The InnoDB engine is fully transactional and supports foreign key references.

The BDB engine, like InnoDB, is also fully transactional and supports foreign key references. However, its use seems to be deprecated.

Other storage engines, including SolidDB and Falcon, are on the horizon. For now, InnoDB is probably your best choice.

1 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.

### 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.

### 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 returns 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 work smoothly with such columns, but if 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) is the recommended solution.

Should you decide to use `utf8_bin` collation for some of your tables with MySQLdb 1.2.1p2, 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. `DATABASE_OPTIONS`.
2. `DATABASE_NAME`, `DATABASE_USER`, `DATABASE_PASSWORD`, `DATABASE_HOST`, `DATABASE_PORT`
3. MySQL option files.

In other words, if you set the name of the database in `DATABASE_OPTIONS`, this will take precedence over `DATABASE_NAME`, which would override anything in a MySQL option file.

Here's a sample configuration which uses a MySQL option file:

```
# settings.py
DATABASE_ENGINE = "mysql"
DATABASE_OPTIONS = {
    'read_default_file': '/path/to/my.cnf',
}

# my.cnf
[client]
database = DATABASE_NAME
user = DATABASE_USER
password = DATABASE_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:

  ```
  DATABASE_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.
- Another method for changing the storage engine is described in AlterModelOnSyncDB.

## Notes on specific fields

### Boolean fields

Since MySQL doesn't have a direct `BOOLEAN` column type, Django uses a `TINYINT` column with values of `1` and `0` to store values for the `BooleanField` model field. Refer to the documentation of that field for more details, but usually this won't be something that will matter unless you're printing out the field values and are expecting to see `True` and `False`..

### 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`.

Furthermore, if you are using a version of MySQL prior to 5.0.3, all of those column types have a maximum length restriction of 255 characters, regardless of whether `unique=True` is specified or not.

## 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.

### String matching for non-ASCII strings

SQLite doesn't support case-insensitive matching for non-ASCII strings. Some possible workarounds for this are documented at sqlite.org, but they are not utilised by the default SQLite backend in Django. Therefore, if you are using the `iexact` lookup type in your queryset filters, be aware that it will not work as expected for non-ASCII strings.

### 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.

However, some platform/Python version combinations include older versions of SQLite (e.g. the official binary distribution of Python 2.5 for Windows, 2.5.4 as of this writing, includes SQLite 3.3.4). There are (as of Django 1.1) even some tests in the Django test suite that will fail when run under this setup.

As described *below*, this can be solved by downloading and installing a newer version of `pysqlite2` (`pysqlite-2.x.x.win32-py2.5.exe` in the described case) that includes and uses a newer version of SQLite. Python 2.6 for Windows ships with a version of SQLite that is not affected by these issues.

## 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

New in version 1.1: *Please, see the release notes*

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:

  ```
  DATABASE_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.

## 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:

```
DATABASE_ENGINE = 'oracle'
DATABASE_NAME = 'xe'
DATABASE_USER = 'a_user'
DATABASE_PASSWORD = 'a_password'
DATABASE_HOST = ''
DATABASE_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 `DATABASE_HOST` and `DATABASE_PORT` like so:

```
DATABASE_ENGINE = 'oracle'
DATABASE_NAME = 'xe'
DATABASE_USER = 'a_user'
DATABASE_PASSWORD = 'a_password'
DATABASE_HOST = 'dbprod01ned.mycompany.com'
DATABASE_PORT = '1540'
```

You should supply both `DATABASE_HOST` and `DATABASE_PORT`, or leave both as empty strings.

### Tablespace options

A common paradigm for optimizing performance in Oracle-based systems is the use of tablespaces to organize disk layout. The Oracle backend supports this use case by adding `db_tablespace` options to the `Meta` and `Field` classes. (When you use a backend that lacks support for tablespaces, Django ignores these options.)

A tablespace can be specified for the table(s) generated by a model by supplying the `db_tablespace` option inside the model's `class Meta`. Additionally, 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 `db_tablespace` option is ignored:

```
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.

New in version 1.0: *Please, see the release notes*

Use the DEFAULT_TABLESPACE and DEFAULT_INDEX_TABLESPACE settings to specify default values for the db_tablespace options. These are useful for setting a tablespace for the built-in Django apps and other applications whose code you cannot control.

Django does not create the tablespaces for you. Please refer to Oracle's documentation for details on creating and managing tablespaces.

## 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.

## 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 coerces the null=True option on fields that have the empty string as a possible value. 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. A workaround to this is to keep TextField columns out of any models that you foresee performing distinct() queries on, and to include the TextField in a related model instead.

## 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

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.

# 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.

## Usage

```
django-admin.py <subcommand> [options]
manage.py <subcommand> [options]
```

`subcommand` should be one of the subcommands listed in this document. `options`, which is optional, should be zero or more of the options available for the given subcommand.

## Getting runtime help

**`--help`**

Run `django-admin.py help` to display a list of all available subcommands. Run `django-admin.py help <subcommand>` to display a description of the given subcommand and a list of its available options.

## App names

Many subcommands 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

**`--version`**

Run `django-admin.py --version` to display the current Django version.

Examples of output:

```
0.95
0.96
0.97-pre-SVN-6069
```

## Displaying debug output

**`--verbosity`** <amount>

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 *default options for django-admin.py*.

## Available subcommands

## cleanup

New in version 1.0: *Please, see the release notes*

Can be run as a cronjob or directly to clean out old data from the database (only expired sessions at the moment).

## *compilemessages*

Changed in version 1.0: Before 1.0 this was the "bin/compile-messages.py" command.

Compiles .po files created with `makemessages` to .mo files for use with the builtin gettext support. See *Internationalization*.

### *--locale*

Use the `--locale` or `-l` option to specify the locale to process. If not provided all locales are processed.

Example usage:

```
django-admin.py compilemessages --locale=br_PT
```

## *createcachetable*

**`django-admin.py createcachetable <tablename>`**

Creates a cache table named `tablename` for use with the database cache backend. See *Django's cache framework* for more information.

## *createsuperuser*

**`django-admin.py createsuperuser`**

New in version 1.0: *Please, see the release notes*

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 e-mail 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.

This command is only available if Django's *authentication system* (`django.contrib.auth`) is installed.

## *dbshell*

**`django-admin.py dbshell`**

Runs the command-line client for the database engine specified in your `DATABASE_ENGINE` setting, with the connection parameters specified in your `DATABASE_USER`, `DATABASE_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.

## *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

**`django-admin.py dumpdata <appname appname appname.Model ...>`**

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.

**`--exclude`**

New in version 1.0: *Please, see the release notes*

Exclude a specific application from the applications whose contents is output. For example, to specifically exclude the *auth* application from the output, 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
```

**`--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.

New in version 1.1: *Please, see the release notes*

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.

## 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.

**`--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.

## inspectdb

Introspects the database tables in the database pointed-to by the `DATABASE_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.

## loaddata <fixture fixture ...>

Searches for and loads the contents of the named fixture into the database.

### 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 and Fixtures

Unfortunately, MySQL isn't capable of completely supporting all the features of Django fixtures. If you use MyISAM tables, MySQL doesn't support transactions or constraints, so you won't get a rollback if multiple transaction files are found, or validation of fixture data. If you use InnoDB tables, you won't be able to have any forward references in your data files - MySQL doesn't provide a mechanism to defer checking of row constraints until a transaction is committed.

## makemessages

Changed in version 1.0: Before 1.0 this was the `bin/make-messages.py` command.

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").

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
```

### --locale

Use the `--locale` or `-l` option to specify the locale to process.

Example usage:

```
django-admin.py makemessages --locale=br_PT
```

### --domain

Use the `--domain` or `-d` option to change the domain of the messages files. Currently supported:

- `django` for all `*.py` and `*.html` files (default)
- `djangojs` for `*.js` files

## reset <appname appname ...>

Executes the equivalent of `sqlreset` for the given app name(s).

### --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.

## *runfcgi [options]*

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.

## *runserver*

**`django-admin.py runserver [port or ipaddr:port]`**

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).

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`.

**`--adminmedia`**

Use the `--adminmedia` option to tell Django where to find the various CSS and JavaScript files for the Django admin interface. Normally, the development server serves these files out of the Django source tree magically, but you'd want to use this if you made any changes to those files for your own site.

Example usage:

```
django-admin.py runserver --adminmedia=/tmp/new-admin-style/
```

**`--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
```

### *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
```

## 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 *How to serve static files*.

## shell

Starts the Python interactive interpreter.

Django will use IPython, if it's installed. If you have IPython installed and want to force use of the "plain" Python interpreter, use the `--plain` option, like so:

```
django-admin.py shell --plain
```

## sql <appname appname ...>

Prints the CREATE TABLE SQL statements for the given app name(s).

## sqlall <appname appname ...>

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.

## sqlclear <appname appname ...>

Prints the DROP TABLE SQL statements for the given app name(s).

## sqlcustom <appname appname ...>

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.

## sqlflush

Prints the SQL statements that would be executed for the flush command.

## sqlindexes <appname appname ...>

Prints the CREATE INDEX SQL statements for the given app name(s).

## sqlreset <appname appname ...>

Prints the DROP TABLE SQL, then the CREATE TABLE SQL, for the given app name(s).

## sqlsequencereset <appname appname ...>

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.

## startapp <appname>

Creates a Django app directory structure for the given app name in the current directory.

## startproject <projectname>

Creates a Django project directory structure for the given project name in the current directory.

This command is disabled when the `--settings` option to `django-admin.py` is used, or when the environment variable `DJANGO_SETTINGS_MODULE` has been set. To re-enable it in these situations, either omit the `--settings` option or unset `DJANGO_SETTINGS_MODULE`.

## 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.

### --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.

## test

Runs tests for all installed models. See *Testing Django applications* for more information.

### --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.

## testserver <fixture fixture ...>

New in version 1.0: *Please, see the release notes*

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` subcommand.

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
```

## validate

Validates all installed models (according to the `INSTALLED_APPS` setting) and prints validation errors to standard output.

# Default options

Although some subcommands may allow their own custom options, every subcommand 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.

# Extra niceties

## Syntax coloring

The `django-admin.py` / `manage.py` commands that output SQL to standard output 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.

## 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.

# File handling reference

## The `File` object

*class* **File** (*file_object*)

## File attributes and methods

Django's `File` has the following attributes and methods:

`File.`**name**
The name of file including the relative path from `MEDIA_ROOT`.

`File.`**path**
The absolute path to the file's location on a local filesystem.

*Custom file storage systems* may not store files locally; files stored on these systems will have a `path` of None.

`File.`**`url`**

The URL where the file can be retrieved. This is often useful in *templates*; for example, a bit of a template for displaying a `Car` (see above) might look like:

```
<img src='{{ car.photo.url }}' alt='{{ car.name }}' />
```

`File.`**`size`**

The size of the file in bytes.

**`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.

**`File.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.

**`File.__iter__`** **()**

Iterate over the file yielding one line at a time.

**`File.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.

**`File.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`.

**`File.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.

**`File.close`** **()**

Close the file.

## Additional `ImageField` attributes

`File.`**`width`**

Width of the image.

`File.`**`height`**

Height of the image.

## Additional methods on files attached to objects

Any `File` that's associated with an object (as with `Car.photo`, above) 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', contents, save=False)
>>> car.save()
```

are the same as this one line:

```
>>> car.photo.save('myphoto.jpg', contents, save=True)
```

Note that the `content` argument must be an instance of `File` or of a subclass of `File`.

**`File.delete (save=True)`**
Remove the file from the model instance and delete the underlying file. The `save` argument works as above.

# File storage API

## Storage.exists(name)

`True` if a file exists given some name.

## Storage.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.

## Storage.size(name)

Returns the total size, in bytes, of the file referenced by `name`.

## Storage.url(name)

Returns the URL where the contents of the file referenced by `name` can be accessed.

## Storage.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.

## Storage.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`.

## Storage.delete(name)

Deletes the file referenced by `name`. This method won't raise an exception if the file doesn't exist.

# Forms

Detailed form API reference. For introductory material, see *Working with forms*.

## 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.

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 e-mail address:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid e-mail 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

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 the cleaned_data attribute of the Form object:

```
>>> 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'}
```

Changed in version 1.0: The cleaned_data attribute was called clean_data in earlier releases.

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 e-mail address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
>>> f.cleaned_data
...
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="subject" maxlength="100" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_message" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name="sender" id="id_sender" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id="id_cc_myself" /></td></tr>
```

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="subject" maxlength="100" value="hello" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_message" value="Hi there" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name="sender" id="id_sender" value="foo@example.com" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id="id_cc_myself" checked="checked" /></td></tr>
```

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()` 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" maxlength="100" /></p>\n<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /></p>\n<p><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></p>\n<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>'
>>> print f.as_p()
<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>
```

## *as_ul()*

`Form.as_ul()` renders the form as a series of `<li>` tags, with each `<li>` containing one field. It does *not* include the `<ul>` or `</ul>`, 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" maxlength="100" /></li>\n<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /></li>\n<li><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></li>\n<li><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_cc_myself" /></li>'
>>> print f.as_ul()
<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text" name="subject" maxlength="100" /></li>
<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" /></li>
<li><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></li>
<li><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_cc_myself" /></li>
```

## *as_table()*

Finally, `Form.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" name="subject" maxlength="100" /></td></tr>\n<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_message" /></td></tr>\n<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name="sender" id="id_sender" /></td></tr>\n<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id="id_cc_myself" /></td></tr>'
>>> print f.as_table()
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject" type="text" name="subject" maxlength="100" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="id_message" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name="sender" id="id_sender" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id="id_cc_myself" /></td></tr>
```

## *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" maxlength="100" /></td></tr>
<tr><th><label for="message">Message:</label></th><td><input type="text" name="message" id="message" /></td></tr>
<tr><th><label for="sender">Sender:</label></th><td><input type="text" name="sender" id="sender" /></td></tr>
<tr><th><label for="cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id="cc_myself" /></td></tr>
>>> print f.as_ul()
<li><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="100" /></li>
<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" /></li>
>>> print f.as_p()
<p><label for="subject">Subject:</label> <input id="subject" type="text" name="subject" maxlength="100" /></p>
<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" /></p>
```

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" name="subject" maxlength="100" /></td></tr>
<tr><th><label for="id_for_message">Message:</label></th><td><input type="text" name="message" id="id_for_message" /></td></tr>
<tr><th><label for="id_for_sender">Sender:</label></th><td><input type="text" name="sender" id="id_for_sender" /></td></tr>
<tr><th><label for="id_for_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself" id="id_for_cc_myself" /></td></tr>
>>> print f.as_ul()
<li><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" name="subject" maxlength="100" /></li>
<li><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_for_message" /></li>
<li><label for="id_for_sender">Sender:</label> <input type="text" name="sender" id="id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_for_cc_myself" /></li>
>>> print f.as_p()
<p><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_for_message">Message:</label> <input type="text" name="message" id="id_for_message" /></p>
<p><label for="id_for_sender">Sender:</label> <input type="text" name="sender" id="id_for_sender" /></p>
<p><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself" id="id_for_cc_myself" /></p>
```

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" maxlength="100" /></li>
<li><label for="id_for_message">Message</label> <input type="text" name="message" id="id_for_message" /></li>
<li><label for="id_for_sender">Sender</label> <input type="text" name="sender" id="id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself</label> <input type="checkbox" name="cc_myself" id="id_for_cc_myself" /></li>
>>> 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" maxlength="100" /></li>
<li><label for="id_for_message">Message -></label> <input type="text" name="message" id="id_for_message" /></li>
<li><label for="id_for_sender">Sender -></label> <input type="text" name="sender" id="id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself -></label> <input type="checkbox" name="cc_myself" id="id_for_cc_myself" /></li>
```

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 e-mail 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" name="subject" maxlength="100" /></td></tr>
<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="text" name="sender" value="invalid e-mail address" /></td></tr>
<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="subject" maxlength="100" /></li>
<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="sender" value="invalid e-mail address" /></li>
<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 e-mail 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 for e in self])
>>> 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 e-mail 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.

To display the HTML for a single field in your form, use dictionary lookup syntax using the field's name as the key, and print the resulting object:

```
>>> f = ContactForm()
>>> print f['subject']
<input id="id_subject" type="text" name="subject" maxlength="100" />
>>> print f['message']
<input type="text" name="message" id="id_message" />
>>> print f['sender']
<input type="text" name="sender" id="id_sender" />
>>> print f['cc_myself']
<input type="checkbox" name="cc_myself" id="id_cc_myself" />
```

Call `str()` or `unicode()` on the field to get its rendered HTML as a string or Unicode object, respectively:

```
>>> str(f['subject'])
'<input id="id_subject" type="text" name="subject" maxlength="100" />'
>>> unicode(f['subject'])
u'<input id="id_subject" type="text" name="subject" maxlength="100" />'
```

Form objects define a custom `__iter__()` method, which allows you to loop through their fields:

```
>>> f = ContactForm()
>>> for field in f: print field
<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. This is 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)
''
```

## Binding uploaded files to a form

New in version 1.0: *Please, see the release notes*

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" id="id_mother-first_name" /></li>
<li><label for="id_mother-last_name">Last name:</label> <input type="text" name="mother-last_name" id="id_mother-last_name" /></li>
>>> print father.as_ul()
<li><label for="id_father-first_name">First name:</label> <input type="text" name="father-first_name" id="id_father-first_name" /></li>
<li><label for="id_father-last_name">Last name:</label> <input type="text" name="father-last_name" id="id_father-last_name" /></li>
```

## 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(u'foo@example.com')
u'foo@example.com'
>>> f.clean('invalid e-mail address')
...
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('')
...
ValidationError: [u'This field is required.']
>>> f.clean(None)
...
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="url" value="http://" /></td></tr>
<tr><th>Comment:</th><td><ul class="errorlist"><li>This field is required.</li></ul><input type="text" name="comment" /></td></tr>
```

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 e-mail 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 />100 characters max.</td></tr>
<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 e-mail address, please.</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" /> 100 characters max.</li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="text" name="sender" /> A valid e-mail 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" /> 100 characters max.</p>
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="text" name="sender" /> A valid e-mail address, please.</p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

### *error_messages*

New in version 1.0: *Please, see the release notes*

`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.

## *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`

Changed in version 1.0: The empty value for a `CheckboxInput` (and hence the standard `BooleanField`) has changed to return `False` instead of `None` in the Django 1.0.

## *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:

`CharField.`**`max_length`**

`CharField.`**`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`

Takes one extra required argument:

`ChoiceField.`**`choices`**

An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field.

## *TypedChoiceField*

*class* **TypedChoiceField** (***kwargs*)

Just like a `ChoiceField`, except `TypedChoiceField` takes an extra `coerce` argument.

- Default widget: `Select`
- Empty value: Whatever you've given as `empty_value`
- Normalizes to: the value returned by the `coerce` argument.
- Validates that the given value exists in the list of choices.
- Error message keys: `required`, `invalid_choice`

Takes extra arguments:

`TypedChoiceField.`**`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.

`TypedChoiceField.`**`empty_value`**

The value to use to represent "empty." Defaults to the empty string; `None` is another common choice here.

## *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:

`DateField.`**`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'
```

Changed in version 1.1: The `DateField` previously used a `TextInput` widget by default. It now uses a `DateInput` widget.

## *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:

`DateTimeField.`**`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'
```

Changed in version 1.0: The `DateTimeField` used to use a `TextInput` widget by default. This has now changed.

## *DecimalField*

New in version 1.0: *Please, see the release notes*

*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`

Takes four optional arguments:

`DecimalField.`**`max_value`**

`DecimalField.`**`min_value`**

These attributes define the limits for the fields value.

`DecimalField.`**`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.

`DecimalField.`**`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 e-mail 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*

New in version 1.0: *Please, see the release notes*

*class* **`FileField`** (***kwargs*)

- Default widget: `FileInput`
- Empty value: `None`
- Normalizes to: An `UploadedFile` object that wraps the file content and file name into a single object.
- Validates that non-empty file data has been bound to the form.
- Error message keys: `required`, `invalid`, `missing`, `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*.

## *FilePathField*

New in version 1.0: *Please, see the release notes*

*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:

`FilePathField.`**`path`**

The absolute path to the directory whose contents you want listed. This directory must exist.

`FilePathField.`**`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.

`FilePathField.`**`match`**
A regular expression pattern; only files with names matching this expression will be allowed as choices.

## *FloatField*

- 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*

New in version 1.0: *Please, see the release notes*

*class* **`ImageField`** (***kwargs*)
- Default widget: `FileInput`
- 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 is installed.

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`

Takes two optional arguments for validation:

`IntegerField.`**`max_value`**

`IntegerField.`**`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`

## *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`

Takes one extra argument, `choices`, as for `ChoiceField`.

## *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:

`RegexField.`**`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.

## *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:

`TimeField.`**`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`, `invalid_link`

Takes the following optional arguments:

`URLField.`**`max_length`**

`URLField.`**`min_length`**

Same as `CharField.max_length` and `CharField.min_length`.

`URLField.`**`verify_exists`**

If `True`, the validator will attempt to load the given URL, raising `ValidationError` if the page gives a 404. Defaults to `False`.

`URLField.`**`validator_user_agent`**

String used as the user-agent used when checking for a URL's existence. Defaults to the value of the `URL_VALIDATOR_USER_AGENT` setting.

## Slightly complex built-in `Field` classes

The following are not yet documented.

*class* **`ComboField`** (***kwargs*)

*class* **`MultiValueField`** (***kwargs*)

*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`

Takes two optional arguments:

`SplitDateTimeField.`**`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.

`SplitDateTimeField.`**`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.

Changed in version 1.1: The `SplitDateTimeField` previously used two `TextInput` widgets by default. The `input_date_formats` and `input_time_formats` arguments are also new.

## Fields which handle relationships

For representing relationships between models, two fields are provided which can derive their choices from a `QuerySet`:

*class* **`ModelChoiceField`** (***kwargs*)

*class* **`ModelMultipleChoiceField`** (***kwargs*)

These fields place one or more model objects into the `cleaned_data` dictionary of forms in which they're used. Both of these fields have an additional required argument:

`ModelChoiceField.`**`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*

Allows the selection of a single model object, suitable for representing a foreign key.

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
```

`ModelChoiceField.`**`empty_label`**

By default the `<select>` widget used by `ModelChoiceField` will have a 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`).

### *ModelMultipleChoiceField*

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.

## 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`).

## 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.

Django provides a representation of all the basic HTML widgets, plus some commonly used groups of widgets:

*class* **`TextInput`**

Text input: `<input type='text' ...>`

*class* **`PasswordInput`**

Password input: `<input type='password' ...>`

*class* **`HiddenInput`**

Hidden input: `<input type='hidden' ...>`

*class* **`MultipleHiddenInput`**

Multiple `<input type='hidden' ...>` widgets.

*class* **`FileInput`**

File upload input: `<input type='file' ...>`

*class* **DateInput**

New in version 1.1: *Please, see the release notes*

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 `'%Y-%m-%d'`.

*class* **DateTimeInput**

New in version 1.0: *Please, see the release notes*

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 `'%Y-%m-%d %H:%M:%S'`.

*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 `'%H:%M:%S'`.

Changed in version 1.1: The `format` argument was not supported in Django 1.0.

*class* **Textarea**

Text area: `<textarea>...</textarea>`

*class* **CheckboxInput**

Checkbox: `<input type='checkbox' ...>`

*class* **Select**

Select widget: `<select><option ...>...</select>`

Requires that your field provides `choices`.

*class* **NullBooleanSelect**

Select widget with options 'Unknown', 'Yes' and 'No'

*class* **SelectMultiple**

Select widget allowing multiple selection: `<select multiple='multiple'>...</select>`

Requires that your field provides `choices`.

*class* **RadioSelect**

A list of radio buttons:

```
<ul>
  <li><input type='radio' ...></li>
  ...
</ul>
```

Requires that your field provides `choices`.

*class* **CheckboxSelectMultiple**

A list of checkboxes:

```
<ul>
  <li><input type='checkbox' ...></li>
  ...
</ul>
```

*class* **MultiWidget**

Wrapper around multiple other widgets

*class* **SplitDateTimeWidget**

Wrapper around two widgets: `DateInput` for the date, and `TimeInput` for the time.

Takes two optional arguments, `date_format` and `time_format`, which work just like the `format` argument for `DateInput` and `TimeInput`.

Changed in version 1.1: The `date_format` and `time_format` arguments were not supported in Django 1.0.

## Specifying widgets

`Form.`**`widget`**

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 for the 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.

## 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:

```
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 `attrs` argument when creating the widget:

`Widget.`**`attrs`**

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>
```

## 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.

The three types of cleaning methods are:

- The `clean()` method on a Field subclass. This is responsible for cleaning the data in a way that is generic for that type of field. For example, a FloatField will turn the data into a Python `float` or raise a `ValidationError`. 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 e-mail 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.

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.

## Form field default cleaning

Let's firstly create a custom form field that validates its input is a string containing comma-separated e-mail addresses, with at least one address. We'll keep it simple and assume e-mail validation is contained in a function called `is_valid_email()`. The full class looks like this:

```
from django import forms

class MultiEmailField(forms.Field):
    def clean(self, value):
        """
        Check that the field contains one or more comma-separated emails
        and normalizes the data to a list of the email strings.
        """
        if not value:
            raise forms.ValidationError('Enter at least one e-mail address.')
        emails = value.split(',')
        for email in emails:
```

```
            if not is_valid_email(email):
                raise forms.ValidationError('%s is not a valid e-mail address.' % email)

        # Always return the cleaned data.
        return emails
```

Every form that uses this field will have this `clean()` method 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.

## *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 way 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 = self.cleaned_data
        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.

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:

```
from django.forms.util import ErrorList

class ContactForm(forms.Form):
    # Everything as before.
    ...

    def clean(self):
        cleaned_data = self.cleaned_data
        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"] = ErrorList([msg])
            self._errors["subject"] = ErrorList([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 behaviour may change in the future, so it's not a bad idea to clean up after yourself in the first place.

# Generic views

Writing Web applications can be monotonous, because we repeat certain patterns again and again. In Django, the most common of these patterns have been abstracted into "generic views" that let you quickly provide common views of an object without actually needing to write any Python code.

A general introduction to generic views can be found in the *topic guide*.

This reference contains details of Django's built-in generic views, along with a list of all keyword arguments that a generic view expects. Remember that arguments may either come from the URL pattern or from the `extra_context` additional-information dictionary.

Most generic views require the `queryset` key, which is a `QuerySet` instance; see *Making queries* for more information about `QuerySet` objects.

# "Simple" generic views

The `django.views.generic.simple` module contains simple views to handle a couple of common cases: rendering a template when no view logic is needed, and issuing a redirect.

## django.views.generic.simple.direct_to_template

**Description:**

Renders a given template, passing it a `{{ params }}` template variable, which is a dictionary of the parameters captured in the URL.

**Required arguments:**

- `template`: The full name of a template to use.

**Optional arguments:**

- `extra_context`: 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 generic view will call it just before rendering the template.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.

**Example:**

Given the following URL patterns:

```
urlpatterns = patterns('django.views.generic.simple',
    (r'^foo/$',             'direct_to_template', {'template': 'foo_index.html'}),
    (r'^foo/(?P<id>\d+)/$', 'direct_to_template', {'template': 'foo_detail.html'}),
)
```

... a request to `/foo/` would render the template `foo_index.html`, and a request to `/foo/15/` would render the `foo_detail.html` with a context variable `{{ params.id }}` that is set to 15.

## django.views.generic.simple.redirect_to

**Description:**

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).

**Required arguments:**

- `url`: The URL to redirect to, as a string. Or `None` to raise a 410 (Gone) HTTP error.

**Optional arguments:**

- `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`.

New in version 1.1: The `permanent` keyword argument is new in Django 1.1.

**Example:**

This example issues a permanent redirect (HTTP status code 301) from `/foo/<id>/` to `/bar/<id>/`:

```
urlpatterns = patterns('django.views.generic.simple',
    ('^foo/(?P<id>\d+)/$', 'redirect_to', {'url': '/bar/%(id)s/'}),
)
```

This example issues a non-permanent redirect (HTTP status code 302) from `/foo/<id>/` to `/bar/<id>/`:

```
urlpatterns = patterns('django.views.generic.simple',
    ('^foo/(?P<id>\d+)/$', 'redirect_to', {'url': '/bar/%(id)s/', 'permanent': False}),
)
```

This example returns a 410 HTTP error for requests to `/bar/`:

```
urlpatterns = patterns('django.views.generic.simple',
    ('^bar/$', 'redirect_to', {'url': None}),
)
```

This example shows how "`%`" characters must be written in the URL in order to avoid confusion with Python's string formatting markers. If the redirect string is written as "`%7Ejacob/`" (with only a single `%`), an exception would be raised:

```
urlpatterns = patterns('django.views.generic.simple',
    ('^bar/$', 'redirect_to', {'url': '%%7Ejacob.'}),
)
```

# Date-based generic views

Date-based generic views (in the module `django.views.generic.date_based`) are views for displaying drilldown pages for date-based data.

## django.views.generic.date_based.archive_index

**Description:**

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`.

**Required arguments:**

- `queryset`: A `QuerySet` of objects for which the archive serves.
- `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.

**Optional arguments:**

- `num_latest`: The number of latest objects to send to the template context. By default, it's 15.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `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`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `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`.

New in version 1.0: *Please, see the release notes*

* template_object_name: Designates the name of the template variable to use in the template context. By default, this is 'latest'.

**Template name:**

If template_name isn't specified, this view will use the template <app_label>/<model_name>_archive.html by default, where:

* <model_name> is your model's name in all lowercase. For a model StaffMember, that'd be staffmember.
* <app_label> is the right-most part of the full Python path to your model's app. For example, if your model lives in apps/blog/models.py, that'd be blog.

**Template context:**

In addition to extra_context, the template's context will be:

* date_list: A list of datetime.date objects representing all years that have objects available according to queryset. These are ordered in reverse. This is equivalent to queryset.dates(date_field, 'year')[::-1].

Changed in version 1.0: The behaviour depending on template_object_name is new in this version.

* latest: The num_latest objects in the system, ordered descending by date_field. For example, if num_latest is 10, then latest will be a list of the latest 10 objects in queryset.

  This variable's name depends on the template_object_name parameter, which is 'latest' by default. If template_object_name is 'foo', this variable's name will be foo.

## *django.views.generic.date_based.archive_year*

**Description:**

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.

**Required arguments:**

* year: The four-digit year for which the archive serves.
* queryset: A QuerySet of objects for which the archive serves.
* 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.

**Optional arguments:**

* template_name: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
* template_loader: The template loader to use when loading the template. By default, it's django.template.loader.
* extra_context: 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 generic view will call it just before rendering the template.
* 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 False.
* context_processors: A list of template-context processors to apply to the view's template.
* template_object_name: Designates the name of the template variable to use in the template context. By default, this is 'object'. The view will append '_list' to the value of this parameter in determining the variable's name.
* 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, this list of objects will be made available to the template as object_list. (The name object_list may be different; see the docs for object_list in the "Template context" section below.) By default, this is False.
* mimetype: The MIME type to use for the resulting document. Defaults to the value of the DEFAULT_CONTENT_TYPE setting.
* 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.

**Template name:**

If template_name isn't specified, this view will use the template <app_label>/<model_name>_archive_year.html by default.

**Template context:**

In addition to extra_context, the template's context will be:

* date_list: A list of datetime.date objects representing all months that have objects available in the given year, according to queryset, in ascending order.
* year: The given year, as a four-character string.

- `object_list`: If the `make_object_list` parameter is `True`, this will be set to a list of objects available for the given year, ordered by the date field. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

  If `make_object_list` is `False`, `object_list` will be passed to the template as an empty list.

## *django.views.generic.date_based.archive_month*

**Description:**

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`.

**Required arguments:**

- `year`: The four-digit year for which the archive serves (a string).
- `month`: The month for which the archive serves, formatted according to the `month_format` argument.
- `queryset`: A `QuerySet` of objects for which the archive serves.
- `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.

**Optional arguments:**

- `month_format`: A format string that regulates what format the `month` parameter uses. This should be in the syntax accepted by Python's `time.strftime`. (See the strftime docs.) It's set to `"%b"` by default, which is a three-letter month abbreviation. To change it to use numbers, use `"%m"`.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `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 `False`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `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`.

**Template name:**

If `template_name` isn't specified, this view will use the template <app_label>/<model_name>_archive_month.html by default.

**Template context:**

In addition to `extra_context`, the template's context will be:

- `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`.
- `object_list`: A list of objects available for the given month. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

## *django.views.generic.date_based.archive_week*

**Description:**

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`.

**Required arguments:**

- `year`: The four-digit year for which the archive serves (a string).
- `week`: The week of the year for which the archive serves (a string). Weeks start with Sunday.
- `queryset`: A `QuerySet` of objects for which the archive serves.

- `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.

**Optional arguments:**

- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `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`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `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`.

**Template name:**

If `template_name` isn't specified, this view will use the template <app_label>/<model_name>_archive_week.html by default.

**Template context:**

In addition to `extra_context`, the template's context will be:

- `week`: A `datetime.date` object representing the first day of the given week.
- `object_list`: A list of objects available for the given week. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

## *django.views.generic.date_based.archive_day*

**Description:**

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`.

**Required arguments:**

- `year`: The four-digit year for which the archive serves (a string).
- `month`: The month for which the archive serves, formatted according to the `month_format` argument.
- `day`: The day for which the archive serves, formatted according to the `day_format` argument.
- `queryset`: A `QuerySet` of objects for which the archive serves.
- `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.

**Optional arguments:**

- `month_format`: A format string that regulates what format the `month` parameter uses. This should be in the syntax accepted by Python's `time.strftime`. (See the strftime docs.) It's set to `"%b"` by default, which is a three-letter month abbreviation. To change it to use numbers, use `"%m"`.
- `day_format`: Like `month_format`, but for the `day` parameter. It defaults to `"%d"` (day of the month as a decimal number, 01-31).
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `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 `False`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `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`.

**Template name:**

If template_name isn't specified, this view will use the template <app_label>/<model_name>_archive_day.html by default.

**Template context:**

In addition to extra_context, 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 given day. Unlike next_day, this will never be None.
- object_list: A list of objects available for the given day. This variable's name depends on the template_object_name parameter, which is 'object' by default. If template_object_name is 'foo', this variable's name will be foo_list.

## *django.views.generic.date_based.archive_today*

**Description:**

A day archive page showing all objects for *today*. This is exactly the same as archive_day, except the year/month/day arguments are not used, and today's date is used instead.

## *django.views.generic.date_based.object_detail*

**Description:**

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.

**Required arguments:**

- year: The object's four-digit year (a string).
- month: The object's month , formatted according to the month_format argument.
- day: The object's day , formatted according to the day_format argument.
- queryset: A QuerySet that contains the object.
- date_field: The name of the DateField or DateTimeField in the QuerySet's model that the generic view should use to look up the object according to year, month and day.
- Either object_id or (slug *and* slug_field) is required.

  If you provide object_id, it should be the value of the primary-key field for the object being displayed on this page.

  Otherwise, slug should be the slug of the given object, and slug_field should be the name of the slug field in the QuerySet's model. By default, slug_field is 'slug'.

**Optional arguments:**

- month_format: A format string that regulates what format the month parameter uses. This should be in the syntax accepted by Python's time.strftime. (See the strftime docs.) It's set to "%b" by default, which is a three-letter month abbreviation. To change it to use numbers, use "%m".
- day_format: Like month_format, but for the day parameter. It defaults to "%d" (day of the month as a decimal number, 01-31).
- template_name: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- template_name_field: The name of a field on the object whose value is the template name to use. This lets you store template names in the data. In other words, if your object has a field 'the_template' that contains a string 'foo.html', and you set template_name_field to 'the_template', then the generic view for this object will use the template 'foo.html'.

  It's a bit of a brain-bender, but it's useful in some cases.
- template_loader: The template loader to use when loading the template. By default, it's django.template.loader.
- extra_context: 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 generic view will call it just before rendering the template.
- context_processors: A list of template-context processors to apply to the view's template.
- template_object_name: Designates the name of the template variable to use in the template context. By default, this is 'object'.
- mimetype: The MIME type to use for the resulting document. Defaults to the value of the DEFAULT_CONTENT_TYPE setting.
- 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.

**Template name:**

If template_name isn't specified, this view will use the template <app_label>/<model_name>_detail.html by default.

**Template context:**

In addition to extra_context, the template's context will be:

- object: The object. This variable's name depends on the template_object_name parameter, which is 'object' by default. If template_object_name is 'foo', this variable's name will be foo.

## List/detail generic views

The list-detail generic-view framework (in the django.views.generic.list_detail module) is similar to the date-based one, except the former simply has two views: a list of objects and an individual object page.

### django.views.generic.list_detail.object_list

**Description:**

A page representing a list of objects.

**Required arguments:**

- queryset: A QuerySet that represents the objects.

**Optional arguments:**

- paginate_by: An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with 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. See Notes on pagination below.
- page: The current page number, as an integer, or the string 'last'. This is 1-based. See Notes on pagination below.
- template_name: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- template_loader: The template loader to use when loading the template. By default, it's django.template.loader.
- extra_context: 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 generic view will call it just before rendering the template.
- 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.
- context_processors: A list of template-context processors to apply to the view's template.
- template_object_name: Designates the name of the template variable to use in the template context. By default, this is 'object'. The view will append '_list' to the value of this parameter in determining the variable's name.
- mimetype: The MIME type to use for the resulting document. Defaults to the value of the DEFAULT_CONTENT_TYPE setting.

**Template name:**

If template_name isn't specified, this view will use the template <app_label>/<model_name>_list.html by default.

**Template context:**

New in version 1.0: The paginator and page_obj context variables are new.

In addition to extra_context, the template's context will be:

- object_list: The list of objects. This variable's name depends on the template_object_name parameter, which is 'object' by default. If template_object_name is 'foo', this variable's name will be foo_list.
- is_paginated: A boolean representing whether the results are paginated. Specifically, this is set to False if the number of available objects is less than or equal to paginate_by.

If the results are paginated, the context will contain these extra variables:

- paginator: An instance of django.core.paginator.Paginator.
- page_obj: An instance of django.core.paginator.Page.

### Notes on pagination

If paginate_by is specified, Django will paginate the results. 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]+)/$', 'object_list', dict(info_dict))
```

- Pass the page number via the page query-string parameter. For example, a URL would look like this:

  ```
  /objects/?page=3
  ```

- To loop over all the available page numbers, use the page_range variable. You can iterate over the list provided by page_range to create a link to every page of results.

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*.

New in version 1.0: *Please, see the release notes*

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.

## django.views.generic.list_detail.object_detail

A page representing an individual object.

**Description:**

A page representing an individual object.

**Required arguments:**

- `queryset`: A `QuerySet` that contains the object.
- Either `object_id` or (`slug` *and* `slug_field`) is required.

  If you provide `object_id`, it should be the value of the primary-key field for the object being displayed on this page.

  Otherwise, `slug` should be the slug of the given object, and `slug_field` should be the name of the slug field in the `QuerySet`'s model. By default, `slug_field` is `'slug'`.

**Optional arguments:**

- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_name_field`: The name of a field on the object whose value is the template name to use. This lets you store template names in the data. In other words, if your object has a field `'the_template'` that contains a string `'foo.html'`, and you set `template_name_field` to `'the_template'`, then the generic view for this object will use the template `'foo.html'`.

  It's a bit of a brain-bender, but it's useful in some cases.
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.

**Template name:**

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_detail.html` by default.

**Template context:**

In addition to `extra_context`, the template's context will be:

- `object`: The object. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

## Create/update/delete generic views

The `django.views.generic.create_update` module contains a set of functions for creating, editing and deleting objects.

Changed in version 1.0: *Please, see the release notes*

`django.views.generic.create_update.create_object` and `django.views.generic.create_update.update_object` now use the new *forms library* to build and display the form.

## `django.views.generic.create_update.create_object`

**Description:**

A page that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.

**Required arguments:**

- Either `form_class` or `model` is required.

  If you provide `form_class`, it should be a `django.forms.ModelForm` subclass. Use this argument when you need to customize the model's form. See the *ModelForm docs* for more information.

  Otherwise, `model` should be a Django model class and the form used will be a standard `ModelForm` for `model`.

**Optional arguments:**

- `post_save_redirect`: A URL to which the view will redirect after saving the object. By default, it's `object.get_absolute_url()`.

  `post_save_redirect` may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use `post_save_redirect="/polls/%(slug)s/"`.
- `login_required`: A boolean that designates whether a user must be logged in, in order to see the page and save changes. This hooks into the Django *authentication system*. By default, this is `False`.

  If this is `True`, and a non-logged-in user attempts to visit this page or save the form, Django will redirect the request to `/accounts/login/`.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.

**Template name:**

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_form.html` by default.

**Template context:**

In addition to `extra_context`, the template's context will be:

- `form`: A `django.forms.ModelForm` instance representing the form for creating the object. This lets you refer to form fields easily in the template system.

  For example, if the model has two fields, `name` and `address`:

  ```
  <form action="" method="post">
  <p>{{ form.name.label_tag }} {{ form.name }}</p>
  <p>{{ form.address.label_tag }} {{ form.address }}</p>
  </form>
  ```

  See the *forms documentation* for more information about using `Form` objects in templates.

## `django.views.generic.create_update.update_object`

**Description:**

A page 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.

**Required arguments:**

- Either `form_class` or `model` is required.

  If you provide `form_class`, it should be a `django.forms.ModelForm` subclass. Use this argument when you need to customize the model's form. See the *ModelForm docs* for more information.

  Otherwise, `model` should be a Django model class and the form used will be a standard `ModelForm` for `model`.
- Either `object_id` or (`slug` *and* `slug_field`) is required.

  If you provide `object_id`, it should be the value of the primary-key field for the object being displayed on this page.

  Otherwise, `slug` should be the slug of the given object, and `slug_field` should be the name of the slug field in the `QuerySet`'s model. By default, `slug_field` is `'slug'`.

**Optional arguments:**

- `post_save_redirect`: A URL to which the view will redirect after saving the object. By default, it's `object.get_absolute_url()`.

  `post_save_redirect` may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use `post_save_redirect="/polls/%(slug)s/"`.
- `login_required`: A boolean that designates whether a user must be logged in, in order to see the page and save changes. This hooks into the Django *authentication system*. By default, this is `False`.

  If this is `True`, and a non-logged-in user attempts to visit this page or save the form, Django will redirect the request to `/accounts/login/`.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`.

**Template name:**

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_form.html` by default.

**Template context:**

In addition to `extra_context`, the template's context will be:

- `form`: A `django.forms.ModelForm` instance representing the form for editing the object. This lets you refer to form fields easily in the template system.

  For example, if the model has two fields, `name` and `address`:

  ```
  <form action="" method="post">
  <p>{{ form.name.label_tag }} {{ form.name }}</p>
  <p>{{ form.address.label_tag }} {{ form.address }}</p>
  </form>
  ```

  See the *forms documentation* for more information about using `Form` objects in templates.
- `object`: The original object being edited. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

## *django.views.generic.create_update.delete_object*

**Description:**

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.

**Required arguments:**

- `model`: The Django model class of the object that the form will create.

- Either `object_id` or (`slug` *and* `slug_field`) is required.

  If you provide `object_id`, it should be the value of the primary-key field for the object being displayed on this page.

  Otherwise, `slug` should be the slug of the given object, and `slug_field` should be the name of the slug field in the `QuerySet`'s model. By default, `slug_field` is `'slug'`.
- `post_delete_redirect`: A URL to which the view will redirect after deleting the object.

**Optional arguments:**

- `login_required`: A boolean that designates whether a user must be logged in, in order to see the page and save changes. This hooks into the Django *authentication system*. By default, this is `False`.

  If this is `True`, and a non-logged-in user attempts to visit this page or save the form, Django will redirect the request to `/accounts/login/`.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: 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 generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`.

**Template name:**

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_confirm_delete.html` by default.

**Template context:**

In addition to `extra_context`, the template's context will be:

- `object`: The original object that's about to be deleted. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

# Built-in middleware reference

This document explains all middleware components that come with Django. For information on how how to use them and how to write your own middleware, see the *middleware usage guide*.

## Available middleware

### Cache middleware

*class* `django.middleware.cache.`**`UpdateCacheMiddleware`**

*class* `django.middleware.cache.`**`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* `django.middleware.common.`**`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/.

  Changed in version 1.0: The behavior of APPEND_SLASH has changed slightly in this version. It didn't used to check whether the pattern was matched in the URLconf.

  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.
- 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* django.middleware.doc.**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.

## GZIP middleware

*class* django.middleware.gzip.**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. Will not compress content bodies less than 200 bytes long, when the response code is something other than 200, JavaScript files (for IE compatibility), or responses that have the Content-Encoding header already specified.

## Conditional GET middleware

*class* django.middleware.http.**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* django.middleware.http.**SetRemoteAddrFromForwardedFor**

This middleware was removed in Django 1.1. See *the release notes* for details.

## Locale middleware

*class* django.middleware.locale.**LocaleMiddleware**

Enables language selection based on data from the request. It customizes content for each user. See the *internationalization documentation*.

## Session middleware

*class* `django.contrib.sessions.middleware.`**`SessionMiddleware`**

Enables session support. See the *session documentation*.

## Authentication middleware

*class* `django.contrib.auth.middleware.`**`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* `django.contrib.csrf.middleware.`**`CsrfMiddleware`**

New in version 1.0: *Please, see the release notes*

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* `django.middleware.transaction.`**`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*.

# Models

Model API reference. For introductory material, see *Models*.

## 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 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 `null=True` option will be coerced for string-based fields that have the empty string as a possible value, and the value `NULL` will be stored to denote the empty string.

## *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):
    GENDER_CHOICES = (
        ('M', 'Male'),
        ('F', 'Female'),
    )
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

or outside your model class altogether:

```
GENDER_CHOICES = (
    ('M', 'Male'),
```

```
    ('F', 'Female'),
)
class Foo(models.Model):
    gender = models.CharField(max_length=1, choices=GENDER_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`**

New in version 1.0: *Please, see the release notes*

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, 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`.

## *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*.

## *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*.

### *BooleanField*

*class* **`BooleanField`** (*`**options`*)

A true/false field.

The admin represents this as a checkbox.

> **MySQL users..**
>
> A boolean field in MySQL is stored as a `TINYINT` column with a value of either 0 or 1 (most databases have a proper `BOOLEAN` type instead). So, for MySQL, only, when a `BooleanField` is retrieved from the database and stored on a model attribute, it will have the values 1 or 0, rather than `True` or `False`. Normally, this shouldn't be a problem, since Python guarantees that `1 == True` and `0 == False` are both true. Just be careful if you're writing something like `obj is True` when `obj` is a value from a boolean attribute on a model. If that model was constructed using the `mysql` backend, the "is" test will fail. Prefer an equality test (using "==") in cases like this.

### *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". The JavaScript calendar will always start the week on a Sunday.

## *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*

New in version 1.0: *Please, see the release notes*

*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

`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).

## EmailField

*class* **EmailField** ([, *max_length=75*, ***options*])

A CharField that checks that the value is a valid e-mail address.

## 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).

Changed in version 1.0: *Please, see the release notes*

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.<br>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**

New in version 1.0: *Please, see the release notes*

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 URL 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 want to retrieve the upload file's on-disk filename, or a URL that refers to that file, or the file's size, you can use the `name`, `url` and `size` attributes; see *Managing files*.

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.

New in version 1.0: The `max_length` argument was added in this version.

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.

## *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.gif`.

`FilePathField.`**`recursive`**
Optional. Either `True` or `False`. Default is `False`. Specifies whether all subdirectories of `path` should be included

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.gif` but not `/home/images/foo/bar.gif` because the `match` applies to the base filename (`foo.gif` and `bar.gif`).

New in version 1.0: The `max_length` argument was added in this version.

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*])

Changed in version 1.0: *Please, see the release notes*

A floating-point number represented in Python by a `float` instance.

The admin represents this as an `<input type="text">` (a single-line input).

## *ImageField*

*class* **ImageField** (*upload_to=None*[, *height_field=None*, *width_field=None*, *max_length=100*, ***options*])

Like `FileField`, but validates that the uploaded object is a valid image. 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.

In addition to the special attributes that are available for `FileField`, an `ImageField` also has `File.height` and `File.width` attributes. See *Managing files*.

Requires the Python Imaging Library.

New in version 1.0: The `max_length` argument was added in this version.

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).

## *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 positive.

## *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** ([, *verify_exists=True*, *max_length=200*, ***options*])

A `CharField` for a URL. Has one extra optional argument:

URLField.**verify_exists**
: If True (the default), the URL given will be checked for existence (i.e., the URL actually loads and doesn't give a 404 response). It should be noted that when using the single-threaded development server, validating a url being serverd by the same server will hang. This should not be a problem for multithreaded servers.

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.

## *XMLField*

*class* **XMLField** (*schema_path=None*[, ***options*])

A `TextField` that checks that the value is valid XML that matches a given schema. Takes one required argument:

**schema_path**
: The filesystem path to a RelaxNG schema against which to validate the field.

## *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):
    # ...
```

New in version 1.0: *Please, see the release notes*

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 (an object with a `get_sql()` method) for more complex queries.
`limit_choices_to` has no effect on the inline FormSets that are created to display related objects in the admin.

`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.

`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.

## *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 names of the two tables being joined. 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` paramter.

`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 two tables being joined.

## 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.

## Related objects reference

This document describes extra methods available on managers when 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`.

**`QuerySet.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.
```

**`QuerySet.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`.

**`QuerySet.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.

**`QuerySet.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`.

# 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 `True`, this model will be an *abstract base class*.

### `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.

### `db_tablespace`

`Options.`**`db_tablespace`**

New in version 1.0: *Please, see the release notes*

The name of the database tablespace to use for the model. 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`**

New in version 1.1: *Please, see the release notes*

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` is `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, a 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'
```

## *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.

> ***Note***
>
> Regardless of how many fields are in `ordering`, the admin site uses only the first field.

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']
```

## *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`**

New in version 1.1: *Please, see the release notes*

If set to `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 list of lists of fields 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).

New in version 1.0: *Please, see the release notes*

For convenience, unique_together can be a single list when dealing with a single set of fields:

```
unique_together = ("driver", "restaurant")
```

### *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"`.

## *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()`.

### *Saving objects*

To save an object back to the database, call `save()`:

**Model.save ([, *force_insert=False*, *force_update=False*])**

Of course, there are some subtleties; see the sections below.

New in version 1.0: *Please, see the release notes*

The signature of the `save()` method has changed from earlier versions (`force_insert` and `force_update` have been added). If you are overriding these methods, be sure to use the correct signature.

### *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. See the documentation for `AutoField` for more details.

## *The pk property*

New in version 1.0: *Please, see the release notes*

`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, `DateFields` 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

New in version 1.0: *Please, see the release notes*

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. Passing both parameters is an error, since 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.

## 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 can be optimized slightly by expressing the update relative to the original field value, rather than as an explicit assignment of a new value. Django provides *F() expressions* as a way of performing this kind of relative update. Using `F()` expressions, the previous example would be 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*.

## Deleting objects

**Model.delete ()**

Issues a SQL DELETE for the object. This only deletes the object in the database; the Python instance will still be around, and will still have data in its fields.

For more details, including how to delete objects in bulk, see *Deleting objects*.

## Other model instance methods

A few object methods have special purposes.

### __str__

**Model.__str__ ()**

`__str__()` is a Python "magic method" that defines what should be returned if you call `str()` on the object. Django uses `str(obj)` (or the related function, `unicode(obj)` -- see below) in a number of places, most notably as the value displayed to render 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 string for the object's `__str__`. Although this isn't required, it's strongly encouraged (see the description of `__unicode__`, below, before putting `__str__` methods everywhere).

For example:

```
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))
```

### __unicode__

**Model.__unicode__ ()**

The `__unicode__()` method is called whenever you call `unicode()` on an object. Since Django's database backends will return Unicode strings in your model's attributes, you would normally want to write a `__unicode__()` method for your model. The example in the previous section could be written more simply as:

```
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.

## *get_absolute_url*

**Model.get_absolute_url ()**

Define a `get_absolute_url()` method to tell Django how to calculate the URL for an object. For example:

```
def get_absolute_url(self):
    return "/people/%i/" % self.id
```

Django uses this in its admin interface. If an object defines `get_absolute_url()`, the object-editing page will have a "View on site" link that will jump you directly to the object's public view, according to `get_absolute_url()`.

Also, a couple of other bits of Django, such as the *syndication feed framework*, use `get_absolute_url()` as a convenience to reward people who've defined the method.

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:

```
<a href="/people/{{ object.id }}/">{{ object.name }}</a>
```

But this template code is good:

```
<a href="{{ object.get_absolute_url }}">{{ object.name }}</a>
```

> ### *Note*
>
> The string you return from `get_absolute_url()` must contain only ASCII characters (required by the URI spec, RFC 2396) that have been URL-encoded, if necessary. Code and templates using `get_absolute_url()` should be able to use the result directly without needing to do 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 a lot.

## *The `permalink` decorator*

The problem with the way we wrote `get_absolute_url()` above is that it slightly violates the DRY principle: the URL for this object is defined both in the URLconf file and in the model.

You can further decouple your models from the URLconf using the `permalink` decorator:

**permalink** ()

This decorator is passed the view function, a list of positional parameters and (optionally) a dictionary of named parameters. Django then works out the correct full URL path using the URLconf, substituting the parameters you have given into the URL. For example, if your URLconf contained 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{1,2})/(?P<day>\d{1,2})/$', 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.month,
        'day': self.created.day})
```

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 tying the model's absolute URL to the view that is used to display it, without repeating the 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 problem, 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.defaults import *

url(r'^people/(\d+)/$',
    'django.views.generic.list_detail.object_detail',
    name='people_view'),
```

...and then using that name to perform the reverse URL resolution instead of the view name:

```
from django.db.models import permalink

def get_absolute_url(self):
    return ('people_view', [str(self.id)])
get_absolute_url = permalink(get_absolute_url)
```

More details on named URL patterns are in the *URL dispatch documentation*.

## Extra instance methods

In addition to `save()`, `delete()`, a model object might get any or all 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, in the following model:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)
class Person(models.Model):
    name = models.CharField(max_length=20)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

...each `Person` instance will have a `get_gender_display()` method. Example:

```
>>> p = Person(name='John', gender='M')
>>> p.save()
>>> p.gender
'M'
>>> p.get_gender_display()
'Male'
```

**`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 the appropriate `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 ID as a fallback check. This guarantees that no records are skipped or duplicated.

# 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, filter, 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. Usually slicing a `QuerySet` returns another (unevaluated) `QuerySet`, but Django will execute the database query if you use the "step" parameter of slice syntax.
- **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 `QuerySets` 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.

## 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.

## 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*])

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.

### QuerySet 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(**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(**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'
OR NOT headline = 'Hello'
```

Note the second example is more restrictive.

#### `annotate(*args, **kwargs)`

New in version 1.1: *Please, see the release notes*

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(*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 the distinct() section 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 ManyToMany 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.

New in version 1.0: *Please, see the release notes*

If you don't want any ordering to be applied to a query, not even the default ordering, call `order_by()` with no parameters.

New in version 1.0: *Please, see the release notes*

The syntax for ordering across related models has changed. See the Django 0.96 documentation for the old behaviour.

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.

New in version 1.1: *Please, see the release notes*

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()

New in version 1.0: *Please, see the release notes*

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()

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(*fields) 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.

## values(*fields)

Returns a `ValuesQuerySet` -- a `QuerySet` that evaluates to a list of dictionaries instead of 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.'}]
```

`values()` 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 couple of subtleties that are worth mentioning:

- The `values()` method does not return anything for `ManyToManyField` attributes and will raise an error if you try to pass in this type of field to it.
- 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 the distinct() section, above, for details.

New in version 1.0: *Please, see the release notes*

Previously, it was not possible to pass `blog_id` to `values()` in the above example, only `blog`.

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.

### *values_list(*fields)*

New in version 1.0: *Please, see the release notes*

This is similar to `values()` except that instead of returning a list of dictionaries, it returns a list of tuples. 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(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)]
```

## *none()*

New in version 1.0: *Please, see the release notes*

Returns an `EmptyQuerySet` -- a `QuerySet` 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()*

New in version 1.0: *Please, see the release notes*

Returns a ''copy'' of the current `QuerySet` (or `QuerySet` subclass you pass in). This can be useful in some situations where you might want to pass in either a model manager or a `QuerySet` and do further filtering on the result. You can safely call `all()` on either object and then you'll definitely have a `QuerySet` to work with.

## *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):
    # ...

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 only refer to `ForeignKey` relations in the list of fields passed to `select_related`. You *can* refer to foreign keys that have `null=True` (unlike the default `select_related()` call). It's an error to use both a list of fields and the `depth` parameter in the same `select_related()` call, since they are conflicting options.

New in version 1.0: *Please, see the release notes*

Both the `depth` argument and the ability to specify field names in the call to `select_related()` are new in Django version 1.0.

### *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:

```
SELECT blog_entry.*, (pub_date > '2006-01-01')
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 entry_count
FROM blog_blog;
```

Note that the parenthesis 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.
New in version 1.0: *Please, see the release notes*
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=['id IN (3, 4, 5, 20)'])
```

...translates (roughly) into the following SQL:

```
SELECT * FROM blog_entry WHERE id IN (3, 4, 5, 20);
```

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(*fields)*

New in version 1.1: *Please, see the release notes*

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 need those particular fields, 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("lede", "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 lede fields.
Entry.objects.defer("body").filter(headline="Lennon").defer("lede")
```

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__lede", "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)
```

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 other models at the same time you shouldn't defer the loading of the field that connects from the primary model to the related one (at the moment, that doesn't raise an error, but it will eventually).

> ### *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. When you are initially developing your applications, don't bother using `defer()`; leave it until your query construction has settled down and you understand where the hot-points are.

## *only(*fields)*

New in version 1.1: *Please, see the release notes*

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 could result in simpler code.

If 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", "lede").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")
```

## *QuerySet 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(**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(**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. So remember to be prepared to handle the exception if you are using manual primary keys.

### *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 behaviour to create() when you are 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.

### *count()*

Returns an integer representing the number of objects in the database matching the QuerySet. count() 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()
```

count() 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.

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(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()*

Evaluates the QuerySet (by performing the query) and returns an iterator over the results. A QuerySet typically reads all of its results and instantiates all of the corresponding objects the first time you access it; iterator() will instead read results and instantiate objects in discrete chunks, yielding them one at a time. For a QuerySet which returns a large number of objects, this often results in better performance and a significant reduction in memory use.

Note that using iterator() on a QuerySet which has already been evaluated will force it to evaluate again, repeating the query.

## 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 an object doesn't exist with the given parameters.

Note `latest()` exists purely for convenience and readability.

## aggregate(*args, **kwargs)

New in version 1.1: *Please, see the release notes*

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 an name 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 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*.

## 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 *Field lookups*.

## 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;
```

Changed in version 1.0: The semantics of `id__exact=None` have changed in Django 1.0. Previously, it was (intentionally) converted to `WHERE id = NULL` at the SQL level, which would never match anything. It has now been changed to behave the same as `id__isnull=True`.

### *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 `'Today Lennon honored'` but not `'today lennon honored'`.

SQLite doesn't support case-sensitive LIKE statements; `contains` acts like `icontains` for SQLite.

## *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%')
```

The above code fragment could also be written as follows:

```
inner_q = Blog.objects.filter(name__contains='Cheddar').values('pk').query
entries = Entry.objects.filter(blog__in=inner_q)
```

Changed in version 1.1: In Django 1.0, only the latter piece of code is valid.

This second form is a bit less readable and unnatural to write, since it accesses the internal `query` attribute and requires a `ValuesQuerySet`. If your code doesn't require compatibility with Django 1.0, use the first form, passing in a queryset directly.

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)
```

## *Warning*

This `query` attribute should be considered an opaque internal attribute. It's fine to use it like above, but its API may change between Django versions.

## *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=values)
```

### *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 doesn't support case-sensitive LIKE statements; `endswith` acts like `iendswith` for SQLite.

### 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 EXTRACT('year' FROM pub_date) = '2005';
```

(The exact SQL syntax varies for each database engine.)

## *month*

For date/datetime fields, 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/datetime fields, 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*

New in version 1.1: *Please, see the release notes*

For date/datetime fields, a 'day of the week' match.

Example:

```
Entry.objects.filter(pub_date__week_day=2)
```

SQL equivalent:

```
SELECT ... WHERE EXTRACT('dow' FROM pub_date) = '2';
```

(The exact SQL syntax varies for each database engine.)

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.

## 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. Please check MySQL documentation for additional details.

## regex

New in version 1.0: *Please, see the release notes*

Case-sensitive regular expression match.

The regular expression syntax is that of the database backend in use. In the case of SQLite, which doesn't natively support regular-expression lookups, the syntax is 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

New in version 1.0: *Please, see the release notes*

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

New in version 1.1: *Please, see the release notes*

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.

- 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: integer

Has one optional argument:

**distinct**
If distinct=True, the count will only include unique instances. This has the SQL equivalent of `COUNT(DISTINCT field)`. 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.

# Request and response objects

## 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.

# HttpRequest objects

*class* **`HttpRequest`**

## Attributes

All attributes except `session` should be considered read-only.

`HttpRequest.`**`path`**

A string representing the full path to the requested page, not including the domain.

Example: `"/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`**

New in version 1.0: *Please, see the release notes*

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` object containing the following attributes:

- `read(num_bytes=None)` -- Read a number of bytes from the file.
- `name` -- The name of the uploaded file.
- `size` -- The size, in bytes, of the uploaded file.
- `chunks(chunk_size=None)` -- A generator that yields sequential chunks of data.

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.

Changed in version 1.0: *Please, see the release notes*

In previous versions of Django, `request.FILES` contained simple `dict` objects representing uploaded files. This is no longer

true -- files are represented by `UploadedFile` objects as described below.
These `UploadedFile` objects will emulate the old-style `dict` interface, but this is deprecated and will be removed in the next release of Django.

`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`
- `CONTENT_TYPE`
- `HTTP_ACCEPT_ENCODING`
- `HTTP_ACCEPT_LANGUAGE`
- `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.

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.`**`raw_post_data`**

The raw HTTP POST data. This is only useful for advanced processing. Use `POST` instead.

`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 ()`**

New in version 1.0: *Please, see the release notes*

Returns the originating host of the request using information from the `HTTP_X_FORWARDED_HOST` 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 333.

Example: `"127.0.0.1:8000"`

**`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`*`)`**

New in version 1.0: *Please, see the release notes*

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.is_secure ()`**

Returns True if the request is secure; that is, if it was made with HTTPS.

**`HttpRequest.is_ajax ()`**

New in version 1.0: *Please, see the release notes*

Returns `True` if the request was made via an `XMLHttpRequest`, by checking the `HTTP_X_REQUESTED_WITH` header for the string `'XMLHttpRequest'`. The following major JavaScript libraries all send this header:

- jQuery
- Dojo
- MochiKit
- MooTools
- Prototype
- YUI

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.

## 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.datastructure.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')
['1', '2']
>>> q['a'] # returns the last
['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()
[('a', '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()
['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*)**

Returns the data with the requested key, as a Python list. Returns an empty list if the key doesn't exist. It's guaranteed to return a list of some sort.

**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()
```

```
[('a', ['1', '2', '3'])]
```

**QueryDict.urlencode ()**

Returns a string of the data in query-string format. Example: "a=2&b=3&b=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:

```
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", mimetype="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>")
```

You can add and delete headers using dictionary syntax:

```
>>> response = HttpResponse()
>>> response['X-DJANGO'] = "It's the best."
>>> del response['X-PHP']
>>> response['X-DJANGO']
"It's the best."
```

Note that del doesn't raise KeyError if the header doesn't exist.

### 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 class:*HttpResponse* instance as a file-like object. Doing so will raise Exception.

### Setting headers

To set a header in your response, just treat it like a dictionary:

```
>>> response = HttpResponse()
>>> response['Pragma'] = 'no-cache'
```

New in version 1.1: *Please, see the release notes*

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 `mimetype` argument and set the `Content-Disposition` header. For example, this is how you might return a Microsoft Excel spreadsheet:

```
>>> response = HttpResponse(my_data, mimetype='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 normal Python string representing the content, encoded from a Unicode object if necessary.

## Methods

**`HttpResponse.__init__ (`*`content=''`*`,`*`mimetype=None`*`,`*`status=200`*`,`*`content_type=DEFAULT_CONTENT_TYPE`*`)`**

Instantiates an `HttpResponse` object with the given page content (a string) and MIME type. The `DEFAULT_CONTENT_TYPE` is `'text/html'`.

`content` can 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.

`status` is the HTTP Status code for the response.

New in version 1.0: *Please, see the release notes*

`content_type` is 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, the `DEFAULT_CONTENT_TYPE` setting is used.

**`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-sensitive.

**`HttpResponse.__getitem__ (`*`header`*`)`**

Returns the value for the given header name. Case-sensitive.

**`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`*`)`**

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.
- `expires` should be a string in the format `"Wdy, DD-Mon-YY HH:MM:SS GMT"`.
- 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.

**`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 URL 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**

New in version 1.0: *Please, see the release notes*

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.

# *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.

## ADMIN_MEDIA_PREFIX

Default: `'/media/'`

The URL prefix for admin media -- CSS, JavaScript and images used by the Django administrative interface. Make sure to use a trailing slash, and to have this be different from the `MEDIA_URL` setting (since the same URL cannot be mapped onto two different sets of files).

## 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 e-mail these people with the full exception information. Each member of the tuple should be a tuple of (Full name, e-mail address). Example:

```
(('John', 'john@example.com'), ('Mary', 'mary@example.com'))
```

Note that Django will e-mail *all* of these people whenever an error happens. See *Error reporting via e-mail* 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`

Whether to append trailing slashes to URLs. This 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*.

## CACHE_BACKEND

Default: `'locmem://'`

The cache backend to use. 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.

## *DATABASE_ENGINE*

Default: `''` (Empty string)

The database backend to use. The built-in database backends are `'postgresql_psycopg2'`, `'postgresql'`, `'mysql'`, `'sqlite3'`, and `'oracle'`.

You can use a database backend that doesn't ship with Django by setting `DATABASE_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.

New in version 1.0: Support for external database backends is new in 1.0.

## *DATABASE_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:

```
DATABASE_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.

## *DATABASE_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`).

## *DATABASE_OPTIONS*

Default: `{}` (Empty dictionary)

Extra parameters to use when connecting to the database. Consult backend module's document for available keywords.

## *DATABASE_PASSWORD*

Default: `''` (Empty string)

The password to use when connecting to the database. Not used with SQLite.

## DATABASE_PORT

Default: `''` (Empty string)

The port to use when connecting to the database. An empty string means the default port. Not used with SQLite.

## DATABASE_USER

Default: `''` (Empty string)

The username to use when connecting to the database. Not used with SQLite.

## DATE_FORMAT

Default: `'N j, Y'` (e.g. `Feb. 4, 2003`)

The default formatting to use for date fields on Django admin change-list pages -- and, possibly, by other parts of the system. See `allowed date format strings`.

See also `DATETIME_FORMAT`, `TIME_FORMAT`, `YEAR_MONTH_FORMAT` and `MONTH_DAY_FORMAT`.

## DATETIME_FORMAT

Default: `'N j, Y, P'` (e.g. `Feb. 4, 2003, 4 p.m.`)

The default formatting to use for datetime fields on Django admin change-list pages -- and, possibly, by other parts of the system. See `allowed date format strings`.

See also `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT`, `YEAR_MONTH_FORMAT` and `MONTH_DAY_FORMAT`.

## DEBUG

Default: `False`

A boolean that turns on/off debug mode.

If you define custom settings, django/views/debug.py has a `HIDDEN_SETTINGS` regular expression which will hide from the DEBUG view anything that contains `'SECRET'`, `'PASSWORD'`, or `'PROFANITIES'`. This allows untrusted users to be able to give backtraces without seeing sensitive (or offensive) settings.

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 are debugging, but on a production server, it will rapidly consume memory.

Never deploy a site into production with `DEBUG` turned on.

## DEBUG_PROPAGATE_EXCEPTIONS

New in version 1.0: *Please, see the release notes*

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.

## 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_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 e-mail address to use for various automated correspondence from the site manager(s).

## DEFAULT_TABLESPACE

New in version 1.0: *Please, see the release notes*

Default: '' (Empty string)

Default tablespace to use for models that don't specify one, if the backend supports it.

## DEFAULT_INDEX_TABLESPACE

New in version 1.0: *Please, see the release notes*

Default: '' (Empty string)

Default tablespace to use for indexes on fields that don't specify one, if the backend supports it.

## 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_HOST

Default: 'localhost'

The host to use for sending e-mail.

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 e-mail 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

New in version 1.0: *Please, see the release notes*

Default: False

Whether to use a TLS (secure) connection when talking to the SMTP server.

## FILE_CHARSET

New in version 1.0: *Please, see the release notes*

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

New in version 1.0: *Please, see the release notes*

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

New in version 1.0: *Please, see the release notes*

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_TEMP_DIR

New in version 1.0: *Please, see the release notes*

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.

## *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.

## *FIXTURE_DIRS*

Default: `()` (Empty tuple)

List of locations of the fixture data files, in search order. Note that these paths should use Unix-style forward slashes, even on Windows. See *Testing Django applications*.

## *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.

## *IGNORABLE_404_ENDS*

Default: `('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi', 'favicon.ico', '.php')`

See also `IGNORABLE_404_STARTS` and `Error reporting via e-mail`.

## *IGNORABLE_404_STARTS*

Default: `('/cgi-bin/', '/_vti_bin', '/_vti_inf')`

A tuple of strings that specify beginnings of URLs that should be ignored by the 404 e-mailer. See `SEND_BROKEN_LINK_EMAILS`, `IGNORABLE_404_ENDS` and the *Error reporting via e-mail*.

## *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`.

## *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*.

## *LANGUAGE_COOKIE_NAME*

New in version 1.0: *Please, see the release notes*

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*.

## *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) -- for example, `('ja', 'Japanese')`. This specifies which languages are available for language selection. See *Internationalization*.

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 displayed 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 *Using translations in your own projects*.

## *LOGIN_REDIRECT_URL*

New in version 1.0: *Please, see the release notes*

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.

## *LOGIN_URL*

New in version 1.0: *Please, see the release notes*

Default: `'/accounts/login/'`

The URL where requests are redirected for login, specially when using the `login_required()` decorator.

## *LOGOUT_URL*

New in version 1.0: *Please, see the release notes*

Default: `'/accounts/logout/'`

LOGIN_URL counterpart.

## *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. Example: `"/home/media/media.lawrence.com/"` See also `MEDIA_URL`.

## *MEDIA_URL*

Default: `''` (Empty string)

URL that handles the media served from `MEDIA_ROOT`. Example: `"http://media.lawrence.com"`

Note that this should have a trailing slash if it has a path component.

Good: `"http://www.example.com/static/"` Bad: `"http://www.example.com/static"`

## *MIDDLEWARE_CLASSES*

Default:

```
('django.middleware.common.CommonMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.contrib.auth.middleware.AuthenticationMiddleware',)
```

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.

## 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

A tuple of profanities, as strings, that will trigger a validation error when the hasNoProfanities validator is called.

We don't list the default values here, because that would be profane. To see the default values, see the file django/conf/global_settings.py.

## 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.

## SEND_BROKEN_LINK_EMAILS

Default: False

Whether to send an e-mail 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_STARTS, IGNORABLE_404_ENDS and *Error reporting via e-mail*.

## 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 e-mail address that error messages come from, such as those sent to ADMINS and MANAGERS.

## SESSION_ENGINE

New in version 1.0: *Please, see the release notes*

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'`

See *How to use sessions*.

## *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 `".lawrence.com"` for cross-domain cookies, or use `None` for a standard domain cookie. See the *How to use sessions*.

## *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*

New in version 1.0: *Please, see the release notes*

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_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*

New in version 1.0: *Please, see the release notes*

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*.

## *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*.

## *TEMPLATE_CONTEXT_PROCESSORS*

Default:

```
("django.core.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media")
```

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.

## *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 `TemplateSyntaxError`. 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, 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.load_template_source',
 'django.template.loaders.app_directories.load_template_source')
```

A tuple of callables (as strings) that know how to import templates from various sources. 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_DATABASE_CHARSET*

New in version 1.0: *Please, see the release notes*

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`, `postgresql_psycopg2`) and MySQL (`mysql`) backends.

## *TEST_DATABASE_COLLATION*

New in version 1.0: *Please, see the release notes*

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 section 10.3.2 of the MySQL manual for details).

## *TEST_DATABASE_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_' + settings.DATABASE_NAME`.

See *Testing Django applications*.

## *TEST_RUNNER*

Default: `'django.test.simple.run_tests'`

The name of the method to use for starting the test suite. See *Testing Django applications*.

## *TIME_FORMAT*

Default: `'P'` (e.g. `4 p.m.`)

The default formatting to use for time fields on Django admin change-list pages -- and, possibly, by other parts of the system. See `allowed date format strings`.

See also `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT`, `YEAR_MONTH_FORMAT` and `MONTH_DAY_FORMAT`.

## *TIME_ZONE*

Default: `'America/Chicago'`

A string representing the time zone for this installation. 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 is the time zone to which Django will convert all dates/times -- not necessarily the timezone of the server. For example, one server may serve multiple Django-powered sites, each with a separate time-zone setting.

Normally, 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 the correct time zone. However, if you're manually *manually configuring settings*, Django will *not* touch the TZ environment variable, and it'll be 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, this variable must be set to match the system timezone.

### URL_VALIDATOR_USER_AGENT

Default: `Django/<version> (http://www.djangoproject.com/)`

The string to use as the `User-Agent` header when checking to see if URLs exist (see the `verify_exists` option on `URLField`).

### USE_ETAGS

Default: `False`

A boolean that specifies whether to output the "Etag" header. This saves bandwidth but slows down performance. This is only used if `CommonMiddleware` is installed (see *Middleware*).

### USE_I18N

Default: `True`

A boolean that specifies whether Django's internationalization 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 internationalization machinery.

### 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`.

## Built-in signal reference

A list of all the signals that Django sends.

> **See also**
>
> The *comment framework* sends a *set of comment-related signals*.

### 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.

## 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 create.

## *pre_delete*

`django.db.models.signals.`**`pre_delete`**

Sent at the beginning of a model's `delete()` method.

Arguments sent with this signal:

**`sender`**
The model class.

**`instance`**
The actual instance being deleted.

## *post_delete*

`django.db.models.signals.`**`post_delete`**

Like `pre_delete`, but sent at the end of the `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.

## *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.

# *Management signals*

Signals sent by *django-admin*.

## *post_syncdb*

`django.db.models.signals.`**`post_syncdb`**

Sent by `syncdb` after it installs an application.

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.

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.

# 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 -- i.e. django.core.handlers.modpython.ModPythonHandler or 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.

# Test signals

Signals only sent when *running tests*.

### *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.

# *Template reference*

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.

## *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*

New in version 1.0: *Please, see the release notes*

Control 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.

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.

#### *block*

Define a block that can be overridden by child templates. See *Template inheritance* for more information.

#### *comment*

Ignore everything between `{% comment %}` and `{% endcomment %}`

#### *cycle*

Changed in version 1.0: Cycle 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 %}
```

Yes, 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:

```
<tr class="{% cycle rowcolors %}">...</tr>
<tr class="{% cycle rowcolors %}">...</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. If you want to escape the variables in the cycle, you must do so explicitly:

```
{% filter force_escape %}
    {% cycle var1 var2 var3 %}
{% endfilter %}
```

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?

## *debug*

Output a whole load of debugging information, including the current context and imported modules.

## *extends*

Signal 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

Filter 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 %}
```

## firstof

Outputs the first variable passed that is not False, without escaping.

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. If you want 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 %}`.

New in version 1.0: *Please, see the release notes*

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*

New in version 1.1: *Please, see the release notes*

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 }}
{% 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 can take an optional `{% else %}` clause that will be displayed if the test fails.

`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 %}
```

`if` tags don't allow and and `or` clauses within the same tag, because the order of logic would be ambiguous. For example, this is invalid:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

If you need to combine and and `or` to do advanced logic, just use nested `if` tags. For example:

```
{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        We have athletes, and either coaches or cheerleaders!
    {% endif %}
{% endif %}
```

Multiple uses of the same logical operator are fine, as long as you use the same operator. For example, this is valid:

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

## *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 a variable, check whether that variable has changed. For example, the following shows the date every time it changes, but only shows the hour if both the hour and 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.

## *ifnotequal*

Just like `ifequal`, except it tests that the two arguments are not equal.

## *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:

  ```
  Hello, {{ person }}
  ```

See also: `{% ssi %}`.

## *load*

Load a custom template tag set.

See *Custom tag and filter libraries* for more information.

## now

Display the date, formatted according to the given string.

Uses the same format as PHP's `date()` function (http://php.net/date) with some custom extensions.

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 match Associated Press style.) | `'a.m.'` |
| A | `'AM'` or `'PM'`. | `'AM'` |
| b | Month, textual, 3 letters, lowercase. | `'jan'` |
| B | Not implemented. | |
| d | Day of the month, 2 digits with leading zeros. | `'01'` to `'31'` |
| D | Day of the week, textual, 3 letters. | `'Fri'` |
| f | Time, in 12-hour hours and minutes, with minutes left off if they're zero. Proprietary extension. | `'1'`, `'1:30'` |
| F | Month, textual, long. | `'January'` |
| g | Hour, 12-hour format without leading zeros. | `'1'` to `'12'` |
| G | Hour, 24-hour format without leading zeros. | `'0'` to `'23'` |
| h | Hour, 12-hour format. | `'01'` to `'12'` |
| H | Hour, 24-hour format. | `'00'` to `'23'` |
| i | Minutes. | `'00'` to `'59'` |
| I | Not implemented. | |
| j | Day of the month without leading zeros. | `'1'` to `'31'` |
| l | Day of the week, textual, long. | `'Friday'` |
| L | Boolean for whether it's a leap year. | `True` or `False` |
| m | Month, 2 digits with leading zeros. | `'01'` to `'12'` |
| M | Month, textual, 3 letters. | `'Jan'` |
| n | Month without leading zeros. | `'1'` to `'12'` |
| N | Month abbreviation in Associated Press style. Proprietary extension. | `'Jan.'`, `'Feb.'`, `'March'`, `'May'` |
| O | Difference to Greenwich time in hours. | `'+0200'` |

| | | |
|---|---|---|
| P | Time, in 12-hour hours, minutes and 'a.m.'/'p.m.', with minutes left off if they're zero and the special-case strings 'midnight' and 'noon' if appropriate. Proprietary extension. | `'1 a.m.'`, `'1:30 p.m.'`, `'midnight'`, `'noon'`, `'12:30 p.m.'` |
| r | RFC 2822 formatted date. | `'Thu, 21 Dec 2000 16:01:07 +0200'` |
| s | Seconds, 2 digits with leading zeros. | `'00'` to `'59'` |
| S | English ordinal suffix for day of the month, 2 characters. | `'st'`, `'nd'`, `'rd'` or `'th'` |
| t | Number of days in the given month. | 28 to 31 |
| T | Time zone of this machine. | `'EST'`, `'MDT'` |
| U | Seconds since the Unix Epoch (January 1 1970 00:00:00 UTC). | |
| w | Day of the week, digits without leading zeros. | `'0'` (Sunday) to `'6'` (Saturday) |
| W | ISO-8601 week number of year, with weeks starting on Monday. | 1, 53 |
| y | Year, 2 digits. | `'99'` |
| Y | Year, 4 digits. | `'1999'` |
| z | Day of the year. | `0` to 365 |
| Z | Time zone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UTC is always positive. | `-43200` to `43200` |

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".

## *regroup*

Regroup a list of alike objects by a common attribute.

This complex tag is best illustrated by use of an example: say that `people` is a list of people represented by dictionaries with `first_name`, `last_name`, and `gender` keys:

```
people = [
    {'first_name': 'George', 'last_name': 'Bush', 'gender': 'Male'},
    {'first_name': 'Bill', 'last_name': 'Clinton', 'gender': 'Male'},
    {'first_name': 'Margaret', 'last_name': 'Thatcher', 'gender': 'Female'},
    {'first_name': 'Condoleezza', 'last_name': 'Rice', 'gender': 'Female'},
    {'first_name': 'Pat', 'last_name': 'Smith', 'gender': 'Unknown'},
]
```

...and you'd like to display a hierarchical list that is ordered by gender, like this:

- **Male:**
    - George Bush
    - Bill Clinton
- **Female:**
    - Margaret Thatcher
    - Condoleezza Rice
- **Unknown:**
    - Pat Smith

You can use the `{% regroup %}` tag to group the list of people by gender. The following snippet of template code would accomplish this:

```
{% regroup people by gender as gender_list %}

<ul>
{% for gender in gender_list %}
    <li>{{ gender.grouper }}
    <ul>
        {% for item in gender.list %}
        <li>{{ item.first_name }} {{ item.last_name }}</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 `people` list by the `gender` attribute and calling the result `gender_list`.

`{% regroup %}` produces a list (in this case, `gender_list`) of **group objects**. Each group object has two attributes:

- `grouper` -- the item that was grouped by (e.g., the string "Male" or "Female").
- `list` -- a list of all items in this group (e.g., a list of all people with gender='Male').

Note that `{% regroup %}` does not order its input! Our example relies on the fact that the `people` list was ordered by `gender` in the first place. If the `people` list did *not* order its members by `gender`, the regrouping would naively display more than one group for a single gender. For example, say the `people` list was set to this (note that the males are not grouped together):

```
people = [
    {'first_name': 'Bill', 'last_name': 'Clinton', 'gender': 'Male'},
    {'first_name': 'Pat', 'last_name': 'Smith', 'gender': 'Unknown'},
    {'first_name': 'Margaret', 'last_name': 'Thatcher', 'gender': 'Female'},
    {'first_name': 'George', 'last_name': 'Bush', 'gender': 'Male'},
    {'first_name': 'Condoleezza', 'last_name': 'Rice', 'gender': 'Female'},
]
```

With this input for `people`, the example `{% regroup %}` template code above would result in the following output:

- **Male:**
    - Bill Clinton
- **Unknown:**
    - Pat Smith
- **Female:**
    - Margaret Thatcher
- **Male:**
    - George Bush
- **Female:**
    - Condoleezza Rice

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 people|dictsort:"gender" by gender as gender_list %}
```

## *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*

Output 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 %}
```

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*

Output 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 URL (i.e., 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 arg1,arg2,name1=value1 %}
```

The first argument is a path to a view function in the format `package.package.module.function`. Additional arguments are optional and should be comma-separated values that will be used as positional and keyword arguments in the URL. 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/`.

New in version 1.0: *Please, see the release notes*

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 `NoReverseMatch` exception raised, which will cause your site to display an error page.

New in version 1.0: *Please, see the release notes*

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 %}
```

New in version 1.1: *Please, see the release notes*

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.

## 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.gif" 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*

New in version 1.0: *Please, see the release notes*

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 business.employees.count as total %}
    {{ total }} employee{{ total|pluralize }}
{% endwith %}
```

The populated variable (in the example above, `total`) is only available between the `{% with %}` and `{% endwith %}` tags.

## *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.

### *addslashes*

Adds slashes before quotes. Useful for escaping strings in CSV, for example.

### *capfirst*

Capitalizes the first character of the value.

### *center*

Centers the value in a field of a given width.

### *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 (same as the now tag).

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'`.

When used without a format string:

```
{{ value|date }}
```

...the formatting string defined in the `DATE_FORMAT` setting will be used.

## *default*

If value evaluates to `False`, use given default. Otherwise, use 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`, use given default. Otherwise, use 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 '
- " (double quote) is converted to "
- & 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.

Changed in version 1.0: Due to auto-escaping, the behavior of this filter has changed slightly. The replacements are only made once, after all other filters are applied -- including filters before and after it.

## *escapejs*

New in version 1.0: *Please, see the release notes*

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.

## *filesizeformat*

Format 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*

Changed in version 1.0: This is rarely useful as ampersands are now 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 & Jerry`.

## *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*

New in version 1.0: *Please, see the release notes*

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.

## *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*

New in version 1.0: *Please, see the release notes*

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 />).

## *linenumbers*

Displays text with line numbers.

## *ljust*

Left-aligns the value in a field of a given width.

**Argument:** field size

## *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 an integer, it's a list of digits. For a string, it's a list of characters.

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 [1, 2, 3].

## phone2numeric

Converts a phone number (possibly containing letters) to its numerical equivalent. For example, '800-COLLECT' will be converted to '800-2655328'.

The input doesn't have to be a valid phone number. This will happily convert any string.

## 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 }}.
```

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" }}.
```

## 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".

## rjust

Right-aligns the value in a field of a given width.

**Argument:** field size

## safe

Marks a string as not requiring further HTML escaping prior to output. When autoescaping is off, this filter has no effect.

## 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.org/native_data_types/lists.html#odbchelper.list.slice for an introduction.

Example:

```
{{ some_list|slice:":2" }}
```

## 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 (same as the now tag). 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".

When used without a format string:

```
{{ value|time }}
```

...the formatting string defined in the TIME_FORMAT setting will be used.

## *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.

## *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 ...".

## *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.

## *unordered_list*

Recursively takes a self-nested list and returns an HTML unordered list -- WITHOUT opening and closing <ul> tags.

Changed in version 1.0: The format accepted by `unordered_list` has changed to be easier to understand.

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: the previous more restrictive and verbose format is still 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.

## urlize

Converts URLs in plain text into clickable links.

Note that if `urlize` is applied to text that already contains HTML markup, things won't work as expected. Apply this filter only to *plain* text.

For example:

```
{{ value|urlize }}
```

If `value` is `"Check out www.djangoproject.com"`, the output will be `"Check out <a href="http://www.djangoproject.com">www.djangoproject.com</a>"`.

## urlizetrunc

Converts URLs into clickable links, truncating URLs longer than the given character limit.

As with urlize, this filter should only be applied to *plain* text.

**Argument:** Length to truncate URLs to

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">www.djangopr...</a>'`.

## wordcount

Returns the number of words.

### 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

Given a string mapping values for true, false and (optionally) None, returns one of those strings according to the value:

| Value | Argument | Outputs |
|---|---|---|
| `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) |

## Other tags and filter 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 (ReStructured Text)

See *markup*.

### django.contrib.webdesign

A collection of template tags that can be useful while designing a website, such as a generator of Lorem Ipsum text. See *django.contrib.webdesign*.

# 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

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

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."
```

Variable names must consist of any letter (A-Z), any digit (0-9), an underscore or a dot.

Dots have a special meaning in template rendering. A dot in a variable name signifies **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`
- Method call. 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."

>>> class PersonClass2:
...     def first_name(self):
...         return "Samantha"
>>> p = PersonClass2()
>>> t.render(Context({"person": p}))
"My name is Samantha."

>>> 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."
```

Method lookups are slightly more complex than the other lookup types. Here are some things to keep in mind:

- If, during the method lookup, a method raises an exception, 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, 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 method call will only work if the method has no required arguments. Otherwise, the system will move to the next lookup type (list-index lookup).
- Obviously, some methods have side effects, 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 a function attribute `alters_data` on the method. The template system won't execute a method if the method has `alters_data=True` set. 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
```

## 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.

## *Playing with Context objects*

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'
```

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
```

Using a `Context` as a stack comes in handy in some custom template tags, as you'll see below.

## *Subclassing Context: 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.core.context_processors.auth",
"django.core.context_processors.debug",
```

```
"django.core.context_processors.i18n",
"django.core.context_processors.media")
```

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.

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.core.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).
- `messages` -- A list of messages (as strings) for the currently logged-in user. Behind the scenes, this calls `request.user.get_and_delete_messages()` for every request. That method collects the user's messages and deletes them from the database.

  Note that messages are set with `user.message_set.create`.
- `perms` -- An instance of `django.core.context_processors.PermWrapper`, representing the permissions that the currently logged-in user has.

### 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* for more.

## django.core.context_processors.media

New in version 1.0: *Please, see the release notes*

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.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.

## 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. Here are the template loaders that come with Django:

**django.template.loaders.filesystem.load_template_source**
Loads templates from the filesystem, according to TEMPLATE_DIRS. This loader is enabled by default.

**django.template.loaders.app_directories.load_template_source**
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 templates in these directories, in this order:

- /path/to/myproject/polls/templates/foo.html
- /path/to/myproject/music/templates/foo.html

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.load_template_source**
Just like app_directories above, but it loads templates from Python eggs rather than from the filesystem.
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

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 -- 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`. All available settings are described in the *settings documentation*, and any setting starting with `TEMPLATE_` is of obvious interest.

#### See also

For information on writing your own custom tags and filters, see *Custom template tags and filters*.

## Unicode data in Django

New in version 1.0: *Please, see the release notes*

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.

### 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 10.3.2 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.

## 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.

### 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 e-mail). 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(u'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.

## 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.)

## 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:

```
qs = People.objects.filter(name__contains=u'Å')
qs = People.objects.filter(name__contains='\xc3\x85') # UTF-8 encoding of Å
```

## Templates

You can use either Unicode or bytestrings when creating templates manually:

```
from django.template import Template
t1 = Template('This is a bytestring template.')
t2 = Template(u'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.

### E-mail

Django's e-mail 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 e-mail specifications, so, for example, e-mail addresses should use only ASCII characters.

The following code example demonstrates that everything except e-mail addresses can be non-ASCII:

```
from django.core.mail import EmailMessage

subject = u'My visit to Sør-Trøndelag'
sender = u'Arnbjörg Ráðormsdóttir <arnbjorg@example.com>'
recipients = ['Fred <fred@example.com']
body = u'...'
EmailMessage(subject, body, sender, recipients).send()
```

### 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.

# 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.

### 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.

## 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.

## 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 e-mail*.
- *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`

## 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.

# 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.

## 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, a core Python principle, 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).

## 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.

## 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.

## 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.

## 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.

## 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.

# *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.

## *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.

# 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 *Generic views*.

**model**
: Models store your application's data.
See *Models*.

**MTV**
: See *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?*.

**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. From the property documentation:

> Properties are a neat way to implement attributes whose usage resembles attribute access, but whose implementation uses method calls. [...] You could only do this by overriding `__getattr__` and `__setattr__`; but overriding `__setattr__` slows down all attribute assignments considerably, and overriding `__getattr__` is always a bit tricky to get right. Properties let you do this painlessly, without having to override `__getattr__` or `__setattr__`.

**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:

```
http://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.

# 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.

## 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

# 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 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:

http://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:

> http://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:

> http://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:

> http://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.

# 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:

> http://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:

> http://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.

# 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:

> http://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:

> http://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.

# 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 bug-fixing 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:

> http://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.

# *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 in Django* 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 monkeypatched 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 |
| 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 http://code.djangoproject.com/wiki/BackwardsIncompatibleChanges.

## 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:

> http://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 in Django*.

## 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*

# *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 Subversion log 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 1.0.2 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 1.1 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 (see the generic views documentation) 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:

- http://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.

# 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:

- http://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.

# 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:

- http://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.

# 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

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 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 features from your code immediately.

`AdminSite.root` will 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 (see the generic views documentation) 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.

## See also

The list of backwards-incompatible changes made in the current development "trunk". If you're running versions of Django newer than an official release, you should keep track of new pieces pointed there. It's also fun reading if you're looking forward to new versions of Django.

# 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.

## *Contributing to Django*

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 many ways you can help Django's development:

- Blog about Django. We syndicate all the Django blogs we know about on the community page; contact jacob@jacobian.org if you've got a blog you'd like to see on that page.
- Report bugs and request features in our ticket tracker. Please read Reporting bugs, below, for the details on how we like our bug reports served up.
- Submit patches for new and/or fixed behavior. Please read Submitting patches, below, for details on how to submit a patch.
- Join the django-developers mailing list and share your ideas for how to improve Django. We're always open to suggestions, although we're likely to be skeptical of large-scale suggestions without some code to back it up.
- Triage patches that have been submitted by other users. Please read Ticket triage below, for details on the triage process.

That's all you need to know if you'd like to join the Django development community. The rest of this document describes the details of how our community works and how it handles bugs, mailing lists, and all the other minutiae of Django development.

### *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** search the tracker to see if your issue has already been filed.
- **Do** ask on django-users *first* if you're not sure if what you're seeing is a bug.
- **Do** write complete, reproducible, specific bug reports. Include as much information as you possibly can, complete with code snippets, test cases, etc. This means including a clear, concise description of the problem, and a clear set of instructions for replicating the problem. A minimal example that illustrates the bug in a nice small test case is the best possible bug report.
- **Don't** use the ticket system to ask support questions. Use the django-users list, or the #django IRC channel for that.
- **Don't** use the ticket system to make large-scale feature requests. We like to discuss any big changes to Django's core on the django-developers list before actually working on them.
- **Don't** reopen issues that have been marked "wontfix". 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 discussion to django-developers.
- **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 triagers, so we see them as they are filed.

### *Reporting security issues*

Report security issues 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.
- Halt all other development as long as is needed to develop a fix, including 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 everyone we know to be running the affected version(s) of Django. We will send these notifications through private e-mail 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.

## 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.

## "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 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 by clicking the radio button next to "Accept ticket" near the bottom of the page, then clicking "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!

Ticket triagers go through the list of claimed tickets from time to time, checking whether any progress has been made. If there's no sign of progress on a particular claimed ticket for a week or two, a triager 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.

## Patch style

- Make sure your code matches our coding style.
- Submit patches in the format returned by the `svn 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.

  Patches in `git diff` format are also acceptable.
- When creating patches, always run `svn diff` from the top-level `trunk` directory -- i.e., the one that contains `django`, `docs`, `tests`, `AUTHORS`, etc. This makes it easy for other people to apply your patches.
- 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.
- 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.
- 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 prevent the problem from arising again).
- If the code associated with a patch adds a new feature, or modifies behavior of an existing feature, the patch should also contain documentation.

## 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.

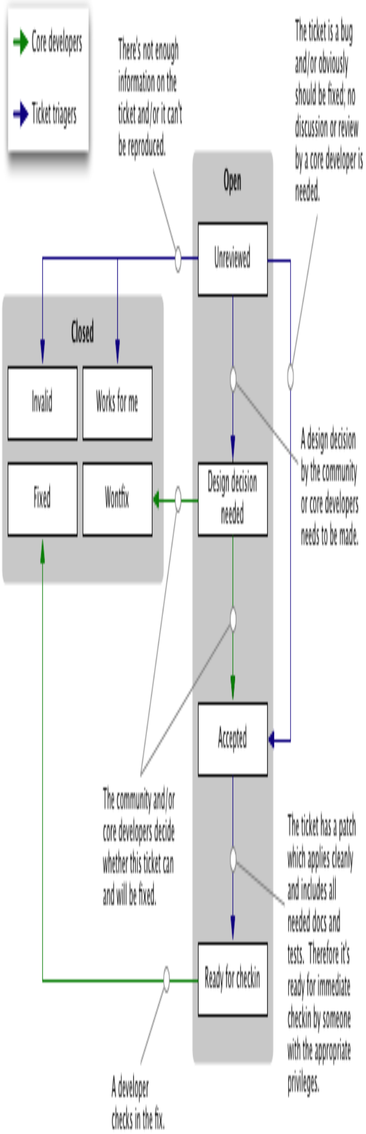
## Ticket triage

Unfortunately, not all bug reports 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* bugs that have been reported by other users. A couple of dedicated volunteers work on this regularly, but more help is always appreciated.

Most of the workflow is based around the concept of a ticket's "triage stage". This stage describes where in its lifetime a given ticket is at any time. Along with a handful of flags, this field 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 official roles here:

- Core developers: people with commit access who make the big decisions and write the bulk of the code.
- Ticket triagers: trusted community members with a proven history of working with the Django community. As a result of this history, they have been entrusted by the core developers to make some of the smaller decisions about tickets.

Second, note the five triage stages:

1. A ticket starts as "Unreviewed", meaning that nobody has examined the ticket.
2. "Design decision needed" means "this concept requires a design decision," which should be discussed either in the ticket comments or on django-developers. The "Design decision needed" step will generally only 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 step and move straight to "Accepted".
3. Once a ticket is ruled to be approved for fixing, it's moved into the "Accepted" stage. This stage is where all the real work gets done.
4. In some cases, a ticket might get moved to the "Someday/Maybe" state. This means the ticket is an enhancement request that we might consider adding to the framework if an excellent patch is submitted. These tickets are not a high priority.
5. If a ticket has an associated patch (see below), a triager will review the patch. If the patch is complete, it'll be marked as "ready for checkin" so that a core developer knows to review and check in the patches.

The second part of this workflow involves a set of flags the describe what the ticket has or needs in order to be "ready for checkin":

**"Has patch"**

This means the ticket has an associated patch. These will be reviewed by the triage team 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 a fix 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, or that the code doesn't live up to our standards.

A ticket can be resolved in a number of ways:

**"fixed"**
: Used by one of 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, and you should feel free to join in when it's something you care about.

**"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.

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 tell us why. Please do not reopen tickets that have been marked as "wontfix" by core developers.

## *Triage by the general community*

Although the core developers and ticket triagers make the big decisions in the ticket triage process, there's also a lot that general community members can do to help the triage process. In particular, 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.
- Correcting the "Needs tests", "Needs documentation", or "Has patch" flags for tickets where they are incorrectly set.
- 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.

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 tickets to "Ready for checkin" unless they are *trivial* changes -- for example, spelling mistakes or broken links in documentation.
- Please **don't** reverse a decision that has been made by a core developer. If you disagree with a discussion that has been made, please post a message to django-developers.
- Please be conservative in your actions. If you're unsure if you should be making a change, don't make the change -- leave a comment with your concerns on the ticket, or post a message to django-developers.

## *Submitting and maintaining translations*

Various parts of Django, such as the admin site and validation error messages, are internationalized. This means they display different text depending on a user's language setting. For this, Django uses the same internationalization infrastructure that is available to Django applications that is described in the *i18n documentation*.

These translations are contributed by Django users worldwide. If you find an incorrect translation, or if you'd like to add a language that isn't yet translated, here's what to do:

- Join the Django i18n mailing list and introduce yourself.
- Create translations using the methods described in the *i18n documentation*. For this you will use the `django-admin.py makemessages` tool. In this particular case it should be run from the top-level `django` directory of the Django source tree.

  The script runs over the entire Django source tree and pulls out all strings marked for translation. It creates (or updates) a message file in the directory `conf/locale` (for example for `pt-BR`, the file will be `conf/locale/pt-br/LC_MESSAGES/django.po`).
- Make sure that `django-admin.py compilemessages -l <lang>` runs without producing any warnings.
- Repeat the last two steps for the `djangojs` domain (by appending the `-d djangojs` command line option to the `django-admin.py` invocations).
- Create a diff of the `.po` file(s) 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.

## Coding style

Please follow these coding standards when writing code for inclusion in Django:

- Unless otherwise specified, follow **PEP 8**.

  You could use a tool like pep8.py 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.
- 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).
- Mark all strings for internationalization; see the *i18n documentation* for details.
- 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
```

- 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.

## 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'),
)
```

## Documentation style

We place a high importance on consistency and readability of documentation. (After all, Django was created in a journalism environment!)

### How to document new features

We treat our documentation like we treat our code: we aim to improve it as often as possible. This section explains how writers can craft their documentation changes in the most useful and least error-prone ways.

Documentation changes 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.

Our policy 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.

There's a full page of information about the *Django documentation system* that you should read prior to working on the documentation.

### Guidelines for ReST files

These guidelines regulate the format of our ReST 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.

### 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.
* **e-mail** -- it has a hyphen.
* **MySQL**
* **PostgreSQL**
* **Python** -- when referring to the language, capitalize Python.
* **realize**, **customize**, **initialize**, etc. -- use the American "ize" suffix, not "ise."
* **SQLite**
* **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.

## Committing code

Please follow these guidelines when committing code to Django's Subversion repository:

- 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."
- For commits to a branch, prefix the commit message with the branch name. For example: "magic-removal: 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.

  Bug fixes need to be added to the current bugfix branch (e.g. the `1.0.X` branch) as well as the current trunk.
- If your commit closes a ticket in the Django ticket tracker, begin your commit message with the text "Fixed #abc", where "abc" is the number of the ticket your commit fixes. Example: "Fixed #123 -- Added support for foo". We've rigged Subversion and 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 #abc." For example: "magic-removal: Fixed #123 -- Added whizbang feature."

  For the curious: We're using a Trac post-commit hook for this.
- If your commit references a ticket in the Django ticket tracker but does *not* close the ticket, include the phrase "Refs #abc", where "abc" is the number of the ticket your commit references. We've rigged Subversion and Trac so that any commit message in that format will automatically post a comment to the appropriate ticket.

## Unit tests

Django comes with a test suite of its own, in the `tests` directory of the Django tarball. 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/<contribapp>/tests`, see below)

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

To run the tests, cd to the `tests/` directory and type:

```
./runtests.py --settings=path.to.django.settings
```

Yes, the unit tests need a settings module, but only for database connection info, with the `DATABASE_ENGINE` setting.

If you're using the `sqlite3` database backend, no further settings are needed. A temporary database will be created in memory when running the tests.

If you're using another backend:

- Your DATABASE_USER setting needs to specify an existing user account for the database engine.
- The DATABASE_NAME setting 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 DATABASE_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_DATABASE_CHARSET in your settings file.

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 the either the python-memcached or cmemcached Python binding

If you want to test the memcached cache backend, you will also need to define a CACHE_BACKEND 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.

To run a subset of the unit tests, append the names of the test modules to the runtests.py command line. See the list of directories in tests/modeltests and tests/regressiontests for module names.

As an example, if Django is not in your PYTHONPATH, you placed settings.py in the tests/ directory, and you'd like to only run tests for generic relations and internationalization, type:

```
PYTHONPATH=..
./runtests.py --settings=settings generic_relations i18n
```

## Contrib apps

Tests for apps in django/contrib/ 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 (it doesn't have to have anything in it). 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
```

## 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 most effectively make a request:

- Request the feature on django-developers, not in the ticket tracker; it'll get read more closely if it's on the mailing list.
- 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.

As with most open-source projects, code talks. If you are willing to write the code for the feature yourself or if (even better) you've already written it, it's much more likely to be accepted. If it's a large feature that might need multiple developers we're always happy to give you an experimental branch in our repository; see below.

## Branch policy

In general, the trunk must be kept stable. People should be able to run production sites against the trunk at any time. Additionally, commits to trunk ought to be as atomic as possible -- smaller changes are better. Thus, large feature changes -- that is, changes too large to be encapsulated in a single patch, or changes that need multiple eyes on them -- must happen on dedicated branches.

This means that if you want to work on a large feature -- anything that would take more than a single patch, or requires large-scale refactoring -- you need to do it on a feature branch. Our development process recognizes two options for feature branches:

1. Feature branches using a distributed revision control system like Git, Mercurial, Bazaar, etc.

   If you're familiar with one of these tools, this is probably your best option since it doesn't require any support or buy-in from the Django core developers.

   However, do keep in mind that Django will continue to use Subversion for the foreseeable future, and this will naturally limit the recognition of your branch. Further, if your branch becomes eligible for merging to trunk you'll need to find a core developer familiar with your DVCS of choice who'll actually perform the merge.

   If you do decided to start a distributed branch of Django and choose to make it public, please add the branch to the Django branches wiki page.
2. Feature branches using SVN have a higher bar. If you want a branch in SVN itself, you'll need a "mentor" among the *core committers*. This person is responsible for actually creating the branch, monitoring your process (see below), and ultimately merging the branch into trunk.

   If you want a feature branch in SVN, you'll need to ask in django-developers for a mentor.

## *Branch rules*

We've got a few rules for branches born out of experience with what makes a successful Django branch.

DVCS branches are obviously not under central control, so we have no way of enforcing these rules. However, if you're using a DVCS, following these rules will give you the best chance of having a successful branch (read: merged back to trunk).

Developers with branches in SVN, however, **must** follow these rules. The branch mentor will keep on eye on the branch and **will delete it** if these rules are broken.

- Only branch entire copies of the Django tree, even if work is only happening on part of that tree. This makes it painless to switch to a branch.
- Merge changes from trunk no less than once a week, and preferably every couple-three days.

  In our experience, doing regular trunk merges is often the difference between a successful branch and one that fizzles and dies.

  If you're working on an SVN branch, you should be using svnmerge.py to track merges from trunk.
- Keep tests passing and documentation up-to-date. As with patches, we'll only merge a branch that comes with tests and documentation.

Once the branch is stable and ready to be merged into the trunk, alert django-developers.

After a branch has been merged, it should be considered "dead"; write access to it will be disabled, and old branches will be periodically "trimmed." To keep our SVN wrangling to a minimum, we won't be merging from a given branch into the trunk more than once.

## *Using branches*

To use a branch, you'll need to do two things:

- Get the branch's code through Subversion.
- Point your Python `site-packages` directory at the branch's version of the `django` package rather than the version you already have installed.

### *Getting the code from Subversion*

To get the latest version of a branch's code, check it out using Subversion:

```
svn co http://code.djangoproject.com/svn/django/branches/<branch>/
```

...where `<branch>` is the branch's name. See the list of branch names.

Alternatively, you can automatically convert an existing directory of the Django source code as long as you've checked it out via Subversion. To do the conversion, execute this command from within your `django` directory:

```
svn switch http://code.djangoproject.com/svn/django/branches/<branch>/
```

The advantage of using `svn switch` instead of `svn co` is that the `switch` command retains any changes you might have made to your local copy of the code. It attempts to merge those changes into the "switched" code. The disadvantage is that it may cause conflicts with your local changes if the "switched" code has altered the same lines of code.

(Note that if you use `svn switch`, you don't need to point Python at the new version, as explained in the next section.)

### Pointing Python at the new Django version

Once you've retrieved the branch's code, you'll need to change your Python `site-packages` directory so that it points to the branch version of the `django` directory. (The `site-packages` directory is somewhere such as `/usr/lib/python2.4/site-packages` or `/usr/local/lib/python2.4/site-packages` or `C:\Python\site-packages`.)

The simplest way to do this is by renaming the old `django` directory to `django.OLD` and moving the trunk version of the code into the directory and calling it `django`.

Alternatively, you can use a symlink called `django` that points to the location of the branch's `django` package. If you want to switch back, just change the symlink to point to the old code.

A third option is to use a path file (`<something>.pth`) which should work on all systems (including Windows, which doesn't have symlinks available). First, make sure there are no files, directories or symlinks named `django` in your `site-packages` directory. Then create a text file named `django.pth` and save it to your `site-packages` directory. That file should contain a path to your copy of Django on a single line and optional comments. Here is an example that points to multiple branches. Just uncomment the line for the branch you want to use ('Trunk' in this example) and make sure all other lines are commented:

```
# Trunk is a svn checkout of:
#   http://code.djangoproject.com/svn/django/trunk/
#
/path/to/trunk

# <branch> is a svn checkout of:
#   http://code.djangoproject.com/svn/django/branches/<branch>/
#
#/path/to/<branch>

# On windows a path may look like this:
# C:/path/to/<branch>
```

If you're using Django 0.95 or earlier and installed it using `python setup.py install`, you'll have a directory called something like `Django-0.95-py2.4.egg` instead of `django`. In this case, edit the file `setuptools.pth` and remove the line that references the Django `.egg` file. Then copy the branch's version of the `django` directory into `site-packages`.

## Deciding on features

Once a feature's been requested and discussed, eventually we'll have a decision about whether to include the feature or drop it.

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. Tough decisions will be discussed by all full committers and finally decided by the Benevolent Dictators for Life, Adrian and Jacob.

## Commit access

Django has two types of committers:

**Full 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 very high for full commit access. It will only be granted by unanimous approval of all existing full committers, and the decision will err on the side of rejection.

**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 subframework to Django and wants to continue to maintain it.

Like full committers, partial commit access is by unanimous approval of all full committers (and any other partial committers in the same area). However, the bar is set lower; proven expertise in the area in question is likely to be sufficient.

To request commit access, please contact an existing committer privately. Public requests for commit access are potential flame-war starters, and will be ignored.

# How the Django documentation works

... and how to contribute.

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 -- `easy_install Sphinx` should do the trick.

Then, building the html is easy; just `make html` 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.

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.

## 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`.

## An example

For a quick example of how it all fits together, check this out:

- First, the ref/settings.txt document starts out like this:

```
.. _ref-settings:

Available settings
==================

...
```

- Next, if you look at the topics/settings.txt document, you can see how a link to ref/settings works:

```
Available settings
==================

For a full list of available settings, see the :ref:`settings reference
<ref-settings>`.
```

- Next, notice how the settings (right now just the top few) 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.

## TODO

The work is mostly done, but here's what's left, in rough order of priority.

- Change the "Added/changed in development version" callouts to proper Sphinx .. versionadded:: or .. versionchanged:: directives.
- Check for and fix malformed links. Do this by running make linkcheck and fix all of the 300+ errors/warnings.

  In particular, look at all the relative links; these need to be changed to proper references.
- 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: there's lots of places that 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.
- Add `.. code-block:: <lang>` to literal blocks so that they get highlighted.

## Hints

Some hints for making things look/read better:

- Whenever possible, use links. So, use ``:setting:`ADMIN_FOR` `` instead of ``` ``ADMIN_FOR`` ```.
- Some directives (`.. setting::`, for one) are prefix-style directives; they go *before* the unit they're describing. These are known as "crossref" directives. Others (`.. class::`, e.g.) 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.
- 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".

# Django committers

## 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 website 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.

## 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 the 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 websites. 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 e-mail 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.

## Specialists

**James Bennett**

James is Django's release manager; he also contributes to the documentation.

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, where he works for the Journal-World developing Ellington. He keeps a blog, has written a book on Django, 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 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.

## 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.

# Django's release process

## Official releases

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 6.0 may break when we release Django 7.0.
- `B` is the *minor version* number, which is incremented for large yet backwards compatible changes. Code written for Django 6.4 will continue to work under Django 6.5.
- `C` is the *micro version* number which, is incremented for bug and security fixes. A new micro-release will always be 100% backwards-compatible with the previous micro-release.
- 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 `N`th alpha/beta/release candidate of version `A.B`.

An exception to this version numbering scheme is the pre-1.0 Django code. There's no guarantee of backwards-compatibility until the 1.0 release.

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 six 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 `PendingDeprecationWarning` 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 always be 100% compatible with the associated minor release -- 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.

# 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.
- All bug fixes applied to the trunk will also be applied to the last minor release, to be released as the next micro release.
- Security fixes will be applied to the current trunk and the previous two minor releases.

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.
- Bug fixes will be applied to a `1.3.X` branch, and released as 1.3.1, 1.3.2, etc.
- Security releases will be applied to trunk, a `1.3.X` branch and a `1.2.X` branch. Security fixes will trigger the release of `1.3.1`, `1.2.1`, etc.

# Release process

Django uses a time-based release schedule, with minor (i.e. 1.1, 1.2, etc.) releases every six months, or more, depending on features.

After each previous release (and after a suitable cooling-off period of a week or two), 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 (i.e 1.1), the previous release will go into bug-fix mode.

A branch will be created of the form `branches/releases/1.0.X` to track bug-fixes to the previous release. When possible, bugs fixed on trunk must *also* be fixed on the bug-fix 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 bug-fix branch. Each bug-fix 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", bug fixes 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.

# Django Deprecation Timeline

This document outlines when various pieces of Django will be removed, following their deprecation, as per the *Django deprecation policy*

- **1.3**
  - `AdminSite.root()`. This release will remove the old method for hooking up admin URLs. This has been deprecated since the 1.1 release.
- **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.

# Indices, glossary and tables

- *Index*
- *Module Index*
- *Glossary*

# Deprecated/obsolete documentation

The following documentation covers features that have been deprecated or that have been replaced in newer versions of Django.

## *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.

### *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).

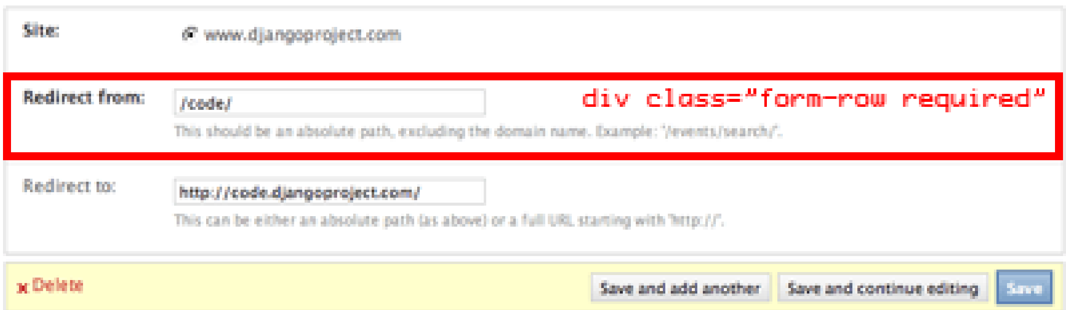
Each fieldset can also take extra classes in addition to `.module` to apply appropriate formatting to the group of fields.

**.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`.

# Index

## Symbols



## _



## A

## D

## *G*



## *H*

## I

## N



## O



## P

## Q



## R

## S

## *U*

# Global Module Index

## D