Jeff Forcier
Paul Bissex
Wesley Chun

Covers
Django 1.0

Python Web Development with Django®

Developer's Library

Python Web Development with Django®

database is legacy or otherwise being used by another application. It's also just a neater way to express refactoring of class definitions without implying an actual object hierarchy.

Let's re-examine (and flesh out) the `Book` and `SmithBook` model hierarchy, using abstract base classes.

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.title

    class Meta:
        abstract = True

class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

The key is the `abstract = True` setting in the `Meta` inner class of `Book`—it signifies that `Book` is an abstract base class and only exists to provide its attributes to the actual model classes which subclass it. Note `SmithBook` only redefines the `authors` field to provide its `limit_choices_to` option—because it inherits from `Book` instead of the usual `models.Model`, the resulting database layout has columns for `title`, `genre`, and `num_pages`, as well as a many-to-many lookup table for `authors`. The Python-level class also has a `__unicode__` method defined as returning the `title` field, just as `Book` does.

In other words, when created in the database, as well as when utilized for object creation, ORM querying, and so forth, `SmithBook` behaves exactly as if it were the following definition:

```
class SmithBook(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })


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

As mentioned, this behavior extends to the query mechanism as well as the attributes of `SmithBook` instances, so the following query would be completely valid:

```
smith_fiction_books = SmithBook.objects.filter(genre='Fiction')
```

Our example isn't fully suited to abstract base classes, however, you'd typically want to create both normal `Book`s as well as `SmithBook`s. Abstract base classes are, of course, abstract—they cannot be created on their own, and as stated previously, are mostly useful to provide DRY at the model definition level. Multi-table inheritance, outlined next, is a better approach for our particular scenario.

Some final notes regarding abstract base classes: The inner `Meta` class on subclasses is inherited from, or combined with, that of the parent class (with the natural exception of the `abstract` option itself, which is reset to `False`, as well as some database-specific options such as `db_name`).

In addition, if a base class uses the `related_name` argument to a relational field such as `ForeignKey`, you need to use some string formatting, so subclasses don't end up clashing. Don't use a normal string, such as `"related_employees"`, but one with `%(class)s` in it, such as `"related_%(class)s"` (refer back to Chapter 1 if you don't recall the details about this type of string replacement). This way, the subclass name is substituted correctly, and collisions are avoided.

Multi-table Inheritance

Multi-table inheritance, at the definition level, appears to be only slightly different from abstract base classes. The use of Python class inheritance is still there, but one simply omits the `abstract = True Meta` class option. When examining model instances, or when querying, multi-table inheritance is again the same as what we've seen before; a subclass appears to inherit all the attributes and methods of its parent class (with the exception of the `Meta` class, as we explain in just a moment).

The primary difference is the underlying mechanism. Parent classes in this scenario are full-fledged Django models with their own database tables and can be instantiated normally as well as lending their attributes to subclasses. This is accomplished by automatically setting up a `OneToOneField` between the subclasses and the parent class, and then performing a bit of behind-the-scenes magic to tie the two objects together, so the subclass inherits the parent class's attributes.

In other words, multi-table inheritance is just a convenience wrapper around a normal "has-a" relationship—or what's known as object composition. Because Django tries to be Pythonic, the "hidden" relationship is actually exposed explicitly if you need it, via the `OneToOneField`, which is given the lowercased name of the parent class with a `_ptr` suffix. For example, in the snippet that follows, `SmithBook` gets a `book_ptr` attribute leading to its "parent" `Book` instance.

The following is our `Book` and `SmithBook` example with multi-table inheritance:

```
class Author(models.Model):
    name = models.CharField(max_length=100)
```

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    num_pages = models.IntegerField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.title

class SmithBook(Book):
    authors = models.ManyToManyField(Author, limit_choices_to={
        'name__endswith': 'Smith'
    })
```

As mentioned, the only difference at this point is the lack of the `Meta` class `abstract` option. Running `manage.py syncdb` on an empty database with this `models.py` file would create three main tables—one each for `Author`, `Book`, and `SmithBook`—whereas with abstract base classes we'd only have tables for `Author` and `SmithBook`.

Note `SmithBook` instances get a `book_ptr` attribute leading back to their composed `Book` instance, and `Book` instances that belong to (or that are part of, depending on how you look at it) `SmithBook`s get a `smithbook` (without a `_ptr` suffix) attribute.

Because this form of inheritance enables the parent class to have its own instances, `Meta` inheritance could cause problems or conflicts between the two sides of the relationship. Therefore, you need to redefine most `Meta` options that can otherwise have been shared between both classes (although `ordering` and `get_latest_by` is inherited if not defined on the child). This makes honoring DRY a little bit tougher, but as much as we'd like to achieve 100 percent DRY, it's not always possible.

Finally, we hope it's relatively clear why this approach is better for our book model; we can instantiate both normal `Book` objects as well as `SmithBook` objects. If you're using model inheritance to map out real-world relationships, chances are you prefer multi-table inheritance instead of abstract base classes. Knowing which approach to use—and when to use neither of them—is a skill that comes with experience.

`Meta` Inner Class

The fields and relationships you define in your models provide the database layout and the variable names you use when querying your model later on—you often find yourself adding model methods such as `__unicode__` and `get_absolute_url` or overriding the built-in `save` or `delete` methods. However, there's a third aspect of model definition and that's the inner class used to inform Django of various metadata concerning the model in question: the `Meta` class.

The `Meta` class, as the name implies, deals with metadata surrounding the model and its use or display: how its name should be displayed when referring to a single object versus multiple objects, what the default sort order should be when querying the database table, the name of that database table (if you have strong opinions on the subject), and so forth.

In addition, the `Meta` class is where you define multi-field uniqueness constraints because it wouldn't make sense to define those inside any single field declaration. Let's add some metadata to our first `Book` example from earlier.

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

    class Meta:
        # Alphabetical order
        ordering = ['title']
```

That's it! The `Book` class is so simple it doesn't need to define most of the options the `Meta` inner class provides, and if we didn't really care about a default ordering, it could have been left out entirely. `Meta` and `Admin` are entirely optional, albeit commonly used, aspects of model definition. Let's whip up a more complex example because `Book`'s meta options are fairly boring.

```
class Person(models.Model):
    first = models.CharField(max_length=100)
    last = models.CharField(max_length=100)
    middle = models.CharField(max_length=100, blank=True)

    class Meta:
    # The proper way to order people, assuming a Last, First M. style of
    # display.
    ordering = ['last', 'first', 'middle']
    # Here we encode the fact that we can't have a person with a 100%
    # identical name. Of course, in real life, we could, but we'll pretend
    # this is an ideal world.
    unique_together = ['first', 'last', 'middle']
    # Django's default pluralization is simply to add 's' to the end: that
    # doesn't work here.
    verbose_name_plural = "people"
```

As you can see from the comments, modeling the concept of a person would be rough going without defining some `Meta` options. We have to consider all three fields when ordering records, and to avoid duplication, and having the system refer to more than one person as "persons" can be quaint, but is probably not desired.

For more details on the various `Meta` class options you can define, we defer you to the official Django documentation.

Admin Registration and Options

If you're using the "admin" contrib app that comes with Django, you are making heavy use of admin `site` objects and their `register` function, as well as optional `ModelAdmin` subclasses. These subclasses enable you to define various options concerning how your model is utilized when you're interacting with it in the admin application.

Simply registering your model class with the admin (along with enabling the Admin app itself, covered in Chapter 2, "Django for the Impatient: Building a Blog") is enough to get the admin to pick it up and provide you with basic list and form pages; hooking in a `ModelAdmin` subclass with extra options enables you to hand-pick the fields displayed in list views, the layout of the forms, and more.

In addition, you can specify inline editing options for relational fields such as `ForeignKey`, by creating `Inline` subclasses and referencing them in a `ModelAdmin` subclass. This proliferation of extra classes can seem odd at first, but it's an extremely flexible way of ensuring any given model can be represented in more than one way or in multiple admin sites. Extending the model hierarchy to inline editing also enables you to place an inline form in more than one "parent" model page, if desired.

We leave the detailed explanation of what each option does to the official documentation—and note there are some examples of admin usage in Part 3—but here's a basic outline of what's possible in each of the two main types of `ModelAdmin` options.

- List formatting: `list_display`, `list_display_links`, `list_filter`, and similar options enable you to change the fields shown in list views (the default being simply the string representation of your model instances in a single column) as well as enabling search fields and filter links, so you can quickly navigate your information.
- Form display: `fields`, `js`, `save_on_top`, and others provide a flexible means of overriding the default form representation of your model, as well as adding custom JavaScript includes and CSS classes, which are useful if you want to try your hand at modifying the look and feel of the admin to fit the rest of your Web site.

Finally, realize if you find yourself making *very* heavy use of these options, it can be a sign to consider disregarding the admin and writing your own administrative forms. However, make sure you read the "Customizing the Admin" section of Chapter 11, "Advanced Django Programming," first for tips on just how much you can flex the Django admin before setting out on your own.

Using Models

Now that we've explained how to define and enhance your models, we go over the details of how to create, and then query, a database based on them, finishing up with notes on the raw SQL underpinnings of the overall mechanism.

Creating and Updating Your Database Using `manage.py`

As mentioned previously in Chapter 2, the `manage.py` script created with every Django project includes functionality for working with your database. The most common `manage.py` command is `syncdb`. Don't let the name fool you; it doesn't do a full synchronization of your database with your models as some users can expect. Instead, it makes sure all model classes are represented as database tables, creating new tables as necessary—but *not* altering existing ones.

> **Database "Synchronization"**
>
> The reasoning behind `syncdb`'s behavior is Django's core development team strongly believes one's production data should never be at the mercy of an automated process. Additionally, it is a commonly held belief that changes to database schemas should only be performed when a developer understands SQL well enough to execute those changes by hand. The authors tend to agree with this approach; a better understanding of underlying technology is always preferable when developing with higher-layer tools.
>
> At the same time, an automatic or semi-automatic schema change-set mechanism (such as Rails' migrations) can often speed up the development process. At the time of this writing, there are several non-core Django-related projects in various stages of development attempting to address this perceived deficit in the framework.

Therefore, if you create a model, run `syncdb` to load it into the database, and later make changes to that model, `syncdb` does not attempt to reconcile those changes with the database. It is expected that the developer makes such changes by hand or via scripts or simply dumps the table or database entirely and reruns `syncdb`, which results in a fully up-to-date schema. For now, what's important is that `syncdb` is the primary method for turning a model class into a database table or tables.

In addition to `syncdb`, `manage.py` provides a handful of specific database-related functions which `syncdb` actually builds upon to perform its own work. Table 4.1 shows a few of the more common ones. Among these are commands such as `sql` and `sqlall`, which display the `CREATE TABLE` statements (`sqlall` performs initial data loading as well); `sqlindexes` for creating indexes; `sqlreset` and `sqlclear`, which empty or drop previously created tables; `sqlcustom`, which executes an app's custom initial SQL statements (see the following for more); and so forth.

Table 4.1 **`manage.py` Functions**

`manage.py` Function	Description
`syncdb`	Create necessary tables needed for all apps
`sql`	Display `CREATE TABLE` call(s)
`sqlall`	Same as `sql` plus initial data-loading statements from `.sql` file
`sqlindexes`	Display the call(s) to create indexes for primary key columns