view.py

def index(request):

next = request.GET.get('next', '/admin')

if request.method == "POST":

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)

data = Students.objects.all()

return render\_to\_response("login/profile.html", {'data', data})

else:

HttpResponse("Inactive User.")

else:

print("User Not Found!")

return HttpResponseRedirect(settings.LOGIN\_URL)

return render(request, 'login/home', {'redirect\_to':next})

adding this in to the views.py

data = Students.objects.all()

stu = {

"student\_number": data

}

return render\_to\_response("login/profile.html", stu)

student module

class Students(models.Model):

student\_number = models.IntegerField(primary\_key=True)

f\_name = models.CharField(max\_length=20, blank=True, null=True)

l\_name = models.CharField(max\_length=20, blank=True, null=True)

dob = models.DateField(blank=True, null=True)

address = models.CharField(max\_length=144, blank=True, null=True)

county = models.CharField(max\_length=20, blank=True, null=True)

phone\_number = models.CharField(max\_length=45, blank=True, null=True)

email = models.CharField(max\_length=45, blank=True, null=True)

gpa = models.IntegerField(blank=True, null=True)

course\_code = models.ForeignKey(Courses, models.DO\_NOTHING, db\_column='course\_code', blank=True, null=True)

college = models.ForeignKey(Colleges, models.DO\_NOTHING, blank=True, null=True)

passwords = models.CharField(max\_length=60, blank=True, null=True)

class Meta:

db\_table = 'students'

html page

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<h1>HEY!!!!</h1>

<p>

{{ data.f\_name}}

{{ data.l\_name }}

{{ data.student\_number }}

{{ data.dob }}

</p>

<a href="/">logout</a>

</body>

</html>

{% for student in student\_number %}

{{ student.f\_name}}

{{ student.l\_name }}

{{ student.student\_number }}

{{ student.dob }}

{% endfor %}

return render(request, 'login/home', {'redirect\_to':next})

Django Models: Basic Data Access

Once you’ve created a model, Django automatically provides a high-level Python API for working with those models. Try it out by running python manage.py shell from within your virtual environment and typing the following:

>>> from books.models import Publisher

>>> p1 = Publisher(name='Apress', address='2855 Telegraph Avenue',

... city='Berkeley', state\_province='CA', country='U.S.A.',

... website='http://www.apress.com/')

>>> p1.save()

>>> p2 = Publisher(name="O'Reilly", address='10 Fawcett St.',

... city='Cambridge', state\_province='MA', country='U.S.A.',

... website='http://www.oreilly.com/')

>>> p2.save()

>>> publisher\_list = Publisher.objects.all()

>>> publisher\_list

<QuerySet [<Publisher: Publisher object>, <Publisher: Publisher object>]>

These few lines of code accomplish quite a bit. Here are the highlights:

* First, we import our Publisher model class. This lets us interact with the database table that contains publishers.
* We create a Publisher object by instantiating it with values for each field – name, address, etc.
* To save the object to the database, call its save() method. Behind the scenes, Django executes an SQL INSERT statement here.
* To retrieve publishers from the database, use the attribute Publisher.objects, which you can think of as a set of all publishers. Fetch a list of all Publisher objects in the database with the statement Publisher.objects.all(). Behind the scenes, Django executes an SQL SELECT statement here.

One thing is worth mentioning, in case it wasn’t clear from this example. When you’re creating objects using the Django model API, Django doesn’t save the objects to the database until you call the save() method:

p1 = Publisher(...)

# At this point, p1 is not saved to the database yet!

p1.save()

# Now it is.

If you want to create an object and save it to the database in a single step, use the objects.create() method. This example is equivalent to the example above:

>>> p1 = Publisher.objects.create(name='Apress',

... address='2855 Telegraph Avenue',

... city='Berkeley', state\_province='CA', country='U.S.A.',

... website='http://www.apress.com/')

>>> p2 = Publisher.objects.create(name="O'Reilly",

... address='10 Fawcett St.', city='Cambridge',

... state\_province='MA', country='U.S.A.',

... website='http://www.oreilly.com/')

>>> publisher\_list = Publisher.objects.all()

>>> publisher\_list

<QuerySet [<Publisher: Publisher object>, <Publisher: Publisher object>]>

Naturally, you can do quite a lot with the Django database API – but first, let’s take care of a small annoyance.

### Adding Model String Representations

When we printed out the list of publishers, all we got was this unhelpful display that makes it difficult to tell the Publisher objects apart:

<QuerySet [<Publisher: Publisher object>, <Publisher: Publisher object>]>

We can fix this easily by adding a method called \_\_str\_\_() to our Publisher class. A \_\_str\_\_()method tells Python how to display a human-readable representation of an object. You can see this in action by adding a \_\_str\_\_() method to the three models:

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 \_\_str\_\_(self):

return self.name

class Author(models.Model):

first\_name = models.CharField(max\_length=30)

last\_name = models.CharField(max\_length=40)

email = models.EmailField()

def \_\_str\_\_(self):

return u'%s %s' % (self.first\_name, self.last\_name)

class Book(models.Model):

title = models.CharField(max\_length=100)

authors = models.ManyToManyField(Author)

publisher = models.ForeignKey(Publisher)

publication\_date = models.DateField()

def \_\_str\_\_(self):

return self.title

As you can see, a \_\_str\_\_() method can do whatever it needs to do in order to return a representation of an object. Here, the \_\_str\_\_() methods for Publisher and Book simply return the object’s name and title, respectively, but the \_\_str\_\_() for Author is slightly more complex – it pieces together the first\_name and last\_name fields, separated by a space. The only requirement for \_\_str\_\_() is that it return a string object. If \_\_str\_\_() doesn’t return a string object – if it returns, say, an integer – then Python will raise a TypeError with a message like:

TypeError: \_\_str\_\_ returned non-string (type int).

For the \_\_str\_\_() changes to take effect, exit out of the Python shell and enter it again with python manage.py shell. (This is the simplest way to make code changes take effect.) Now the list of Publisher objects is much easier to understand:

>>> from books.models import Publisher

>>> publisher\_list = Publisher.objects.all()

>>> publisher\_list

<QuerySet [<Publisher: Apress>, <Publisher: O'Reilly>]>

Make sure any model you define has a \_\_str\_\_() method – not only for your own convenience when using the interactive interpreter, but also because Django uses the output of \_\_str\_\_() in several places when it needs to display objects. Finally, note that \_\_str\_\_() is a good example of adding behavior to models. A Django model describes more than the database table layout for an object; it also describes any functionality that object knows how to do. \_\_str\_\_() is one example of such functionality – a model knows how to display itself.

### Inserting and Updating Data

You’ve already seen this done: to insert a row into your database, first create an instance of your model using keyword arguments, like so:

>>> p = Publisher(name='GNW Independent Publishing',

... address='123 Some Street',

... city='Hamilton',

... state\_province='NSW',

... country='AUSTRALIA',

... website='http://djangobook.com/')

As we noted above, this act of instantiating a model class does not touch the database. The record isn’t saved into the database until you call save(), like this:

>>> p.save()

In SQL, this can roughly be translated into the following:

INSERT INTO books\_publisher

(name, address, city, state\_province, country, website)

VALUES

('GNW Independent Publishing', '123 Some Street', 'Hamilton', 'NSW',

'Australia', 'http://djangobook.com/');

Because the Publisher model uses an auto incrementing primary key id, the initial call to save()does one more thing: it calculates the primary key value for the record and sets it to the idattribute on the instance:

>>> p.id

3 # this will differ based on your own data

Subsequent calls to save() will save the record in place, without creating a new record (i.e., performing an SQL UPDATE statement instead of an INSERT):

>>> p.name = 'GNW Independent Publishing'

>>> p.save()

The preceding save() statement will result in roughly the following SQL:

UPDATE books\_publisher SET

name = 'GNW Independent Publishing',

address='123 Some Street',

city='Hamilton',

state\_province='NSW',

country='AUSTRALIA',

website='http://djangobook.com/')

WHERE id = 3;

Yes, note that all of the fields will be updated, not just the ones that have been changed. Depending on your application, this may cause a race condition. See “Updating Multiple Objects in One Statement” below to find out how to execute this (slightly different) query:

UPDATE books\_publisher SET

name = 'GNW Independent Publishing'

WHERE id=3;

### Selecting Objects

Knowing how to create and update database records is essential, but chances are that the web applications you’ll build will be doing more querying of existing objects than creating new ones. We’ve already seen a way to retrieve every record for a given model:

>>> Publisher.objects.all()

<QuerySet [<Publisher: Apress>, <Publisher: O'Reilly>, <Publisher: GNW Independent Publishing>]>

This roughly translates to this SQL:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher;

Notice that Django doesn’t use SELECT \* when looking up data and instead lists all fields explicitly. This is by design: in certain circumstances SELECT \* can be slower, and (more important) listing fields more closely follows one tenet of the Zen of Python: “Explicit is better than implicit.” For more on the Zen of Python, try typing import thisat a Python prompt.

Let’s take a close look at each part of this Publisher.objects.all() line:

* First, we have the model we defined, Publisher. No surprise here: when you want to look up data, you use the model for that data.
* Next, we have the objects attribute. This is called a manager. Managers are discussed in detail in Chapter 9. For now, all you need to know is that managers take care of all table-level operations on data including, most important, data lookup. All models automatically get an objects manager; you’ll use it any time you want to look up model instances.
* Finally, we have all(). This is a method on the objects manager that returns all the rows in the database in a QuerySet – an object that represents a specific set of rows from the database. Appendix C deals with QuerySets in detail.

Any database lookup is going to follow this general pattern – we’ll call methods on the manager attached to the model we want to query against.

### Filtering Data

Naturally, it’s rare to want to select everything from a database at once; in most cases, you’ll want to deal with a subset of your data. In the Django API, you can filter your data using the filter()method:

>>> Publisher.objects.filter(name='Apress')

<QuerySet [<Publisher: Apress>]>

filter() takes keyword arguments that get translated into the appropriate SQL WHERE clauses. The preceding example would get translated into something like this:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

WHERE name = 'Apress';

You can pass multiple arguments into filter() to narrow down things further:

>>> Publisher.objects.filter(country="U.S.A.",

state\_province="CA")

<QuerySet [<Publisher: Apress>]>

Those multiple arguments get translated into SQL AND clauses. Thus, the example in the code snippet translates into the following:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

WHERE country = 'U.S.A.'

AND state\_province = 'CA';

Notice that by default the lookups use the SQL = operator to do exact match lookups. Other lookup types are available:

>>> Publisher.objects.filter(name\_\_contains="press")

<QuerySet [<Publisher: Apress>]>

That’s a double underscore there between name and contains. Like Python itself, Django uses the double underscore to signal that something “magic” is happening – here, the \_\_contains part gets translated by Django into a SQL LIKE statement:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

WHERE name LIKE '%press%';

Many other types of lookups are available, including icontains (case-insensitive LIKE), startswith and endswith, and range (SQL BETWEEN queries). Appendix C describes all of these lookup types in detail.

### Retrieving Single Objects

The filter() examples above all returned a QuerySet, which you can treat like a list. Sometimes it’s more convenient to fetch only a single object, as opposed to a QuerySet. That’s what the get()method is for:

>>> Publisher.objects.get(name="Apress")

<Publisher: Apress>

Instead of a QuerySet, only a single object is returned. Because of that, a query resulting in multiple objects will cause an exception:

>>> Publisher.objects.get(country="U.S.A.")

Traceback (most recent call last):

...

books.models.MultipleObjectsReturned: get() returned more than one Publisher - it returned 2!

A query that returns no objects also causes an exception:

>>> Publisher.objects.get(name="Penguin")

Traceback (most recent call last):

...

books.models.DoesNotExist: Publisher matching query does not exist.

The DoesNotExist exception is an attribute of the model’s class – Publisher.DoesNotExist. In your applications, you’ll want to trap these exceptions, like this:

try:

p = Publisher.objects.get(name='Apress')

except Publisher.DoesNotExist:

print ("Apress isn't in the database yet.")

else:

print ("Apress is in the database.")

### Ordering Data

As you play around with the previous examples, you might discover that the objects are being returned in a seemingly random order. You aren’t imagining things; so far we haven’t told the database how to order its results, so we’re simply getting back data in some arbitrary order chosen by the database. In your Django applications, you’ll probably want to order your results according to a certain value – say, alphabetically. To do this, use the order\_by() method:

>>> Publisher.objects.order\_by("name")

<QuerySet [<Publisher: Apress>, <Publisher: GNW Independent Publishing>, <Publisher: O'Reilly>]>

This doesn’t look much different from the earlier all() example, but the SQL now includes a specific ordering:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

ORDER BY name;

You can order by any field you like:

>>> Publisher.objects.order\_by("address")

<QuerySet [<Publisher: O'Reilly>, <Publisher: GNW Independent Publishing>, <Publisher: Apress>]>

>>> Publisher.objects.order\_by("state\_province")

<QuerySet [<Publisher: Apress>, <Publisher: O'Reilly>, <Publisher: GNW Independent Publishing>]>

To order by multiple fields (where the second field is used to disambiguate ordering in cases where the first is the same), use multiple arguments:

>>> Publisher.objects.order\_by("state\_province", "address")

<QuerySet [<Publisher: Apress>, <Publisher: O'Reilly>, <Publisher: GNW Independent Publishing>]>

You can also specify reverse ordering by prefixing the field name with a “-” (that’s a minus character):

>>> Publisher.objects.order\_by("-name")

<QuerySet [<Publisher: O'Reilly>, <Publisher: GNW Independent Publishing>, <Publisher: Apress>]>

While this flexibility is useful, using order\_by() all the time can be quite repetitive. Most of the time you’ll have a particular field you usually want to order by. In these cases, Django lets you specify a default ordering in the model:

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 \_\_str\_\_(self):

return self.name

class Meta:

ordering = ['name']

Here, I’ve introduced a new concept: the class Meta, which is a class that’s embedded within the Publisher class definition (i.e., it’s indented to be within class Publisher). You can use this Metaclass on any model to specify various model-specific options. A full reference of Meta options is available in Appendix B, but for now, we’re concerned with the ordering option. If you specify this, it tells Django that unless an ordering is given explicitly with order\_by(), all Publisher objects should be ordered by the name field whenever they’re retrieved with the Django database API.

### Chaining Lookups

You’ve seen how you can filter data, and you’ve seen how you can order it. Often, of course, you’ll need to do both. In these cases, you simply “chain” the lookups together:

>>> Publisher.objects.filter(country="U.S.A.").order\_by("-name")

<QuerySet [<Publisher: O'Reilly>, <Publisher: Apress>]>

As you might expect, this translates to a SQL query with both a WHERE and an ORDER BY:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

WHERE country = 'U.S.A'

ORDER BY name DESC;

### Slicing Data

Another common need is to look up only a fixed number of rows. Imagine you have thousands of publishers in your database, but you want to display only the first one. Thankfully, a Django QuerySet can be treated just like a Python list. This means you can do this using Python’s standard list slicing syntax:

>>> Publisher.objects.order\_by('name')[0]

<Publisher: Apress>

This translates roughly to:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

ORDER BY name

LIMIT 1;

Similarly, you can retrieve a specific subset of data using Python’s range-slicing syntax:

>>> Publisher.objects.order\_by('name')[0:2]

This returns two objects, translating roughly to:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

ORDER BY name

OFFSET 0 LIMIT 2;

Note that negative slicing is not supported:

>>> Publisher.objects.order\_by('name')[-1]

Traceback (most recent call last):

...

AssertionError: Negative indexing is not supported.

This is easy to get around, though. Just change the order\_by() statement, like this:

>>> Publisher.objects.order\_by('-name')[0]

### Updating Multiple Objects in One Statement

I pointed out in the “Inserting and Updating Data” section that the model save() method updates all columns in a row. Depending on your application, you may want to update only a subset of columns. For example, let’s say we want to update the Apress Publisher to change the name from 'Apress' to 'Apress Publishing'. Using save(), it would look something like this:

>>> p = Publisher.objects.get(name='Apress')

>>> p.name = 'Apress Publishing'

>>> p.save()

This roughly translates to the following SQL:

SELECT id, name, address, city, state\_province, country, website

FROM books\_publisher

WHERE name = 'Apress';

UPDATE books\_publisher SET

name = 'Apress Publishing',

address = '2855 Telegraph Ave.',

city = 'Berkeley',

state\_province = 'CA',

country = 'U.S.A.',

website = 'http://www.apress.com'

WHERE id = 1;

(Note that this example assumes Apress has a publisher ID of 1.) You can see in this example that Django’s save() method sets all of the column values, not just the name column. If you’re in an environment where other columns of the database might change due to some other process, it’s smarter to change only the column you need to change. To do this, use the update() method on QuerySet objects. Here’s an example:

>>> Publisher.objects.filter(id=1).update(name='Apress Publishing')

The SQL translation here is much more efficient and has no chance of race conditions:

UPDATE books\_publisher

SET name = 'Apress Publishing'

WHERE id = 1;

The update() method works on any QuerySet, which means you can edit multiple records in bulk. Here’s how you might change the country from 'U.S.A.' to USA in each Publisher record:

>>> Publisher.objects.all().update(country='USA')

3

The update() method has a return value – an integer representing how many records changed. In the above example, we got 3.

### Deleting Objects

To delete an object from your database, simply call the object’s delete() method:

>>> p = Publisher.objects.get(name="O'Reilly")

>>> p.delete()

(1, {'books.Publisher': 1})

>>> Publisher.objects.all()

<QuerySet [<Publisher: Apress>, <Publisher: GNW Independent Publishing>]>

Note the return value from Django when you delete an object – Django first lists the total number of records that will be affected (in this case one) and a dictionary containing each of the models (tables) affected and how many records were deleted in each table.

You can also delete objects in bulk by calling delete() on the result of any QuerySet. This is similar to the update() method we showed in the last section:

>>> Publisher.objects.filter(country='USA').delete()

(1, {'books.Publisher': 1})

>>> Publisher.objects.all().delete()

(1, {'books.Publisher': 1})

>>> Publisher.objects.all()

<QuerySet []>

Be careful deleting your data! As a precaution against deleting all of the data in a particular table, Django requires you to explicitly use all() if you want to delete everything in your table. For example, this won’t work:

>>> Publisher.objects.delete()

Traceback (most recent call last):

File "", line 1, in

AttributeError: 'Manager' object has no attribute 'delete'

But it’ll work if you add the all() method:

>>> Publisher.objects.all().delete()

Note, If you’re just deleting a subset of your data, you don’t need to include all(). To repeat a previous example:

>>> Publisher.objects.filter(country='USA').delete()