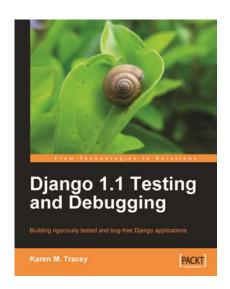


Django 1.1 Testing and Debugging

Karen M. Tracey



Chapter No.3 "Testing 1, 2, 3: Basic Unit Testing"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Testing 1, 2, 3: Basic Unit Testing"

A synopsis of the book's content

Information on where to buy this book

About the Author

Karen has a PhD in Electrical/Computer Engineering from the University of Notre Dame. Her research there focused on distributed operating systems, which led to work in an industry centered on communications protocols and middleware. Outside of work she has an interest in puzzles, which led her to take up crossword construction. She has published nearly 100 puzzles in the New York Times, the Los Angeles Times syndicate, the New York Sun, and USA Today. She amassed a database of thousands of puzzles to aid in constructing and cluing her own puzzles. The desire to put a web frontend on this database is what led her to Django. She was impressed by the framework and its community, and became an active core framework contributor. Karen is one of the most prolific posters on the django-users mailing list. Her experience in helping hundreds of people there guided her in choosing the best and most useful material to include in this book.

For More Information:

Many thanks to Steven Wilding and the entire Packt Publishing team for making this book possible.

I'd also like to thank the Django community. The community is too large to name everyone individually, but Jacob Kaplan-Moss, Adrian Holovaty, Malcolm Tredinnick, and Russell Keith-Magee deserve special mention. I very much appreciate the tremendous amount of work you all have done to create an excellent framework and foster a helpful and welcoming community.

Finally thanks to my parents, brothers, and many friends who supported me throughout the writing process. Your encouraging

words have been very helpful and much appreciated.

For More Information:

Django 1.1 Testing and Debugging

Bugs are a time consuming burden during software development. Django's built-in test framework and debugging support help lessen this burden. This book will teach you quick and efficient techniques for using Django and Python tools to eradicate

bugs and ensure your Django application works correctly. This book will walk you stepby-step through the development of a complete sample

Django application. You will learn how best to test and debug models, views, URL configuration, templates, and template tags. This book will help you integrate with and make use of the rich external environment of testing and debugging tools for Python and Django applications.

This book starts with a basic overview of testing. It will highlight areas to look out for while testing. You will learn about the different kinds of tests available, the pros and cons of each, and details of test extensions provided by Django that simplify the task of testing Django applications. You will see an illustration of how external tools that provide even more sophisticated testing features can be integrated into Django's framework.

On the debugging front, the book illustrates how to interpret the extensive debugging information provided by Django's debug error pages, and how to utilize logging and other external tools to learn what code is doing.

This book is a step-by-step guide to running tests using Django's test support and making best use of Django and Python debugging tools.

What This Book Covers

In *Chapter 1, Django Testing Overview*, we begin development of a sample Django survey application. The example tests automatically generated by Django are described and run. All of the options available for running tests are covered.

In *Chapter 2, Does This Code Work? Doctests in Depth*, the models used by the sample application are developed. Using doctests to test models is illustrated by example. The pros and cons of doctests are discussed. Specific caveats for using doctests with Django applications are presented.'

In *Chapter 3, Testing 1, 2, 3: Basic Unit Testing*, the doctests implemented in the previous chapter are re-implemented as unit tests and assessed in light of the pros, cons, and caveats of doctests discussed in the previous chapter. Additional tests are developed that need to make use of test data. Using fixture files to load such data is demonstrated. In addition, some tests where fixture files are inappropriate for test data are developed.

For More Information:

In Chapter 4, Getting Fancier: Django Unit Test Extensions, we begin to write the views that serve up web pages for the application. The number of tests is starting to become significant, so this chapter begins by showing how to replace use of a single tests.py fi le for tests with a tests directory, so that tests may be kept well-organized. Then, tests for views are developed that illustrate how unit test extensions provided by Django simplify the task of testing web applications. Testing form behavior is demonstrated by development of a test for an admin customization made in this chapter.

Chapter 5, Filling in the Blanks: Integrating Django and Other Test Tools, shows how Django supports integration of other test tools into its framework. Two examples are presented. The first illustrates how an add-on application can be used to generate test coverage information while the second demonstrates how use of the twill test tool (which allows for much easier testing of form behavior) can be integrated into Django application tests.

Chapter 6, Django Debugging Overview, provides an introduction to the topic of debugging Django applications. All of the settings relevant for debugging are described. Debug error pages are introduced. The database query history maintained by Django when debugging is turned on is described, as well as features of the development server that aid in debugging. Finally, the handling of errors that occur during production (when debug is off) is detailed, and all the settings necessary to ensure that information about such errors is captured and sent to the appropriate people are mentioned

In Chapter 7, When the Wheels Fall Off: Understanding a Django Debug Page, development of the sample application continues, making some typical mistakes along the way. These mistakes result in Django debug pages. All of the information available on these pages is described, and guidance on what pieces are likely most helpful to look at in what situations is given. Several different kinds of debug pages are encountered and discussed in depth.

Chapter 8, When Problems Hide: Getting More Information, focuses on how to get more information about how code is behaving in cases where a problem doesn't result in a debug error page. It walks through the development of a template tag to embed the query history for a view in the rendered page, and then shows how the Django debug toolbar can be used to get the same information, in addition to much more. Finally, some logging utilities are developed.

Chapter 9, When You Don't Even Know What to Log: Using Debuggers, walks through examples of using the Python debugger (pdb) to track down what is going wrong in cases where no debug page appears and even logging isn't helpful. All of the most useful pdb commands are illustrated by example. In addition, we see how pdb can be used to ensure correct code behavior for code that is subject to multi-process race conditions.

Chapter 10, When All Else Fails: Getting Outside Help, describes what to do when none of the techniques covered so far have solved a problem. Possibly, it is a bug in external

For More Information:

code: tips are given on how to search to see if others have experienced the same and if there are any fixes available. Possibly it's a bug in our code or a misunderstanding about how some things work; avenues for asking questions and tips on writing good questions are included.

In Chapter 11, When it's Time to Go Live: Moving to Production, we move the sample application into production, using Apache and mod_wsgi instead of the development server. Several of the most common problems encountered during this step are covered. In addition, the option of using Apache with mod_wsgi during development is discussed.

For More Information:

Testing 1, 2, 3: Basic Unit Testing

In the previous chapter, we began learning about testing Django applications by writing some doctests for the Survey model. In the process, we experienced some of the advantages and disadvantages of doctests. When discussing some of the disadvantages, unit tests were mentioned as an alternative test approach that avoids some doctest pitfalls. In this chapter, we will start to learn about unit tests in detail. Specifically, we will:

- Re-implement the Survey doctests as unit tests
- Assess how the equivalent unit test version compares to the doctests in terms
 of ease of implementation and susceptibility to the doctest caveats discussed
 in the previous chapter
- Begin learning some of the additional capabilities of unit tests as we extend the existing tests to cover additional functions

Unit tests for the Survey save override method

Recall in the previous chapter that we ultimately implemented four individual tests of the Survey save override function:

- A straightforward test of the added capability, which verifies that if closes is not specified when a Survey is created, it is auto-set to a week after opens
- A test that verifies that this auto-set operation is not performed if closes is explicitly specified during creation

For More Information:

Testing 1, 2, 3: Basic Unit Testing

- A test that verifies that closes is only auto-set if its value is missing during initial creation, not while saving an existing instance
- A test that verifies that the save override function does not introduce an unexpected exception in the error case where neither opens nor closes is specified during creation

To implement these as unit tests instead of doctests, create a TestCase within the suvery/tests.py file, replacing the sample SimpleTest. Within the new TestCase class, define each individual test as a separate test method in that TestCase, like so:

```
import datetime
from django.test import TestCase
from django.db import IntegrityError
from survey.models import Survey
class SurveySaveTest(TestCase):
    t = "New Year's Resolutions"
    sd = datetime.date(2009, 12, 28)
    def testClosesAutoset(self):
        s = Survey.objects.create(title=self.t, opens=self.sd)
        self.assertEqual(s.closes, datetime.date(2010, 1, 4))
    def testClosesHonored(self):
        s = Survey.objects.create(title=self.t, opens=self.sd,
                                  closes=self.sd)
        self.assertEqual(s.closes, self.sd)
    def testClosesReset(self):
        s = Survey.objects.create(title=self.t, opens=self.sd)
        s.closes = None
        self.assertRaises(IntegrityError, s.save)
    def testTitleOnly(self):
        self.assertRaises(IntegrityError, Survey.objects.create,
                          title=self.t)
```

This is more difficult to implement than the doctest version, isn't it? It is not possible to use a direct cut-and-paste from a shell session, and there is a fair amount of code overhead—code that does not appear anywhere in the shell session—that needs to be added. We can still use cut-and-paste from our shell session as a starting point, but we must edit the code after pasting it, in order to turn the pasted code into a proper unit test. Though not difficult, this can be tedious.

Chapter 3

Most of the extra work consists of choosing names for the individual test methods, minor editing of cut-and-pasted code to refer to class variables such as t and sd correctly, and creating the appropriate test assertions to verify the expected result. The first of these requires the most brainpower (choosing good names can be hard), the second is trivial, and the third is fairly mechanical. For example, in a shell session where we had:

```
datetime.date(2010, 1, 4)
>>>
In the unit test, we instead have an assertEqual:
    self.assertEqual(s.closes, datetime.date(2010, 1, 4))
Expected exceptions are similar, but use assertRaises. For example, where in a shell session we had:
>>> s = Survey.objects.create(title=t)
Traceback (most recent call last):
    [ traceback details snipped ]
IntegrityError: survey_survey.opens may not be NULL
>>>
In the unit test, this is:
    self.assertRaises(IntegrityError, Survey.objects.create, title=self.t)
```

Note we do not actually call the create routine in our unit test code, but rather leave that up to the code within assertRaises. The first parameter passed to assertRaises is the expected exception, followed by the callable expected to raise the exception, followed by any parameters that need to be passed to the callable when calling it.

Pros of the unit test version

>>> s.closes

What do we get from this additional work? Right off the bat, we get a little more feedback from the test runner, when running at the highest verbosity level. For the doctest version, the output of manage.py test survey -v2 was simply:

```
Doctest: survey.models.Survey.save ... ok
```

Testing 1, 2, 3: Basic Unit Testing

For the unit test version, we get individual results reported for each test method:

```
testClosesAutoset (survey.tests.SurveySaveTest) ... ok
testClosesHonored (survey.tests.SurveySaveTest) ... ok
testClosesReset (survey.tests.SurveySaveTest) ... ok
testTitleOnly (survey.tests.SurveySaveTest) ... ok
```

If we take a little more effort and provide single-line docstrings for our test methods, we can get even more descriptive results from the test runner. For example, if we add docstrings like so:

```
class SurveySaveTest(TestCase):
   """Tests for the Survey save override method"""
   t = "New Year's Resolutions"
   sd = datetime.date(2009, 12, 28)
   def testClosesAutoset(self):
        """Verify closes is autoset correctly"""
        s = Survey.objects.create(title=self.t, opens=self.sd)
        self.assertEqual(s.closes, datetime.date(2010, 1, 4))
   def testClosesHonored(self):
        """Verify closes is honored if specified"""
       s = Survey.objects.create(title=self.t, opens=self.sd,
                                  closes=self.sd)
       self.assertEqual(s.closes, self.sd)
   def testClosesReset(self):
        """Verify closes is only autoset during initial create"""
        s = Survey.objects.create(title=self.t, opens=self.sd)
       s.closes = None
       self.assertRaises(IntegrityError, s.save)
   def testTitleOnly(self):
        """Verify correct exception is raised in error case"""
       self.assertRaises(IntegrityError, Survey.objects.create,
                           title=self.t)
```

The test runner output for this test will then be:

```
Verify closes is autoset correctly ... ok
Verify closes is honored if specified ... ok
Verify closes is only autoset during initial create ... ok
Verify correct exception is raised in error case ... ok
```

Buy Django 1.1 Testing and Debugging ebook with Django 1.0 Website Development
ebook and get 50% off both. Just add both the eBooks to your cart. Next, add
'djg11wed'in the Promotion Code field and then click the "Add Promotion Code"
button, the discount will be applied. This offer is valid till 31st May 2010. Grab your
copy now!!!

α		,
(na	pter	
Cim	$\rho \iota \iota \iota \iota$	4

This additional descriptive detail may not be that important when all tests pass, but when they fail, it can be very helpful as a clue to what the test is trying to accomplish.

For example, let's assume we have broken the save override method by neglecting to add seven days to opens, so that if closes is not specified, it is auto-set to the same

```
value as opens. With the doctest version of the test, the failure would be reported as:
FAIL: Doctest: survey.models.Survey.save
______
Traceback (most recent call last):
  File "/usr/lib/python2.5/site-packages/django/test/ doctest.py", line
2180, in runTest
   raise self.failureException(self.format_failure(new.getvalue()))
AssertionError: Failed doctest test for survey.models.Survey.save
  File "/dj projects/marketr/survey/models.py", line 10, in save
File "/dj projects/marketr/survey/models.py", line 19, in survey.models.
Survey.save
Failed example:
   s.closes
Expected:
   datetime.date(2010, 1, 4)
Got:
   datetime.date(2009, 12, 28)
That doesn't give much information on what has gone wrong, and you really
have to go read the full test code to see what is even being tested. The same failure
reported by the unit test is a bit more descriptive, as the FAIL header includes the test
docstring, so we immediately know the problem has something to do with closes
being auto-set:
______
FAIL: Verify closes is autoset correctly
_____
Traceback (most recent call last):
  File "/dj_projects/marketr/survey/tests.py", line 20, in
testClosesAutoset
   self.assertEqual(s.closes, datetime.date(2010, 1, 4))
AssertionError: datetime.date(2009, 12, 28) != datetime.date(2010, 1, 4)
                               - [ 67 ] ·
```

Testing 1, 2, 3: Basic Unit Testing

We can take this one step further and make the error message a bit friendlier by specifying our own error message on the call to assertEqual:

```
def testClosesAutoset(self):
    """Verify closes is autoset correctly"""
    s = Survey.objects.create(title=self.t, opens=self.sd)
    self.assertEqual(s.closes, datetime.date(2010, 1, 4),
        "closes not autoset to 7 days after opens, expected %s, "
        "actually %s" %
        (datetime.date(2010, 1, 4), s.closes))
```

The reported failure would then be:

```
FAIL: Verify closes is autoset correctly

Traceback (most recent call last):

File "/dj_projects/marketr/survey/tests.py", line 22, in testClosesAutoset

(datetime.date(2010, 1, 4), s.closes))

AssertionError: closes not autoset to 7 days after opens, expected 2010-01-04, actually 2009-12-28
```

In this case, the custom error message may not be much more useful than the default one, since what the save override is supposed to do here is quite simple. However, such custom error messages can be valuable for more complicated test assertions to help explain what is being tested, and the "why" behind the expected result.

Another benefit of unit tests is that they allow for more selective test execution than doctests. On the manage.py test command line, one or more unit tests to be executed can be identified by TestCase name. You can even specify that only particular methods in a TestCase should be run. For example:

```
python manage.py test survey.SurveySaveTest.testClosesAutoset
```

Here we are indicating that we only want to run the testClosesAutoset test method in the SurveySaveTest unit test found in the survey application. Being able to run just a single method or a single test case is a very convenient time saver when developing tests.

Chapter 3

Cons of the unit test version

Has anything been lost by switching to unit tests? A bit. First, there is the ease of implementation that has already been mentioned: unit tests require more work to implement than doctests. Though generally not difficult work, it can be tedious. It is also work where errors can be made, resulting in a need to debug the test code. This increased implementation burden can serve to discourage writing comprehensive tests.

We've also lost the nice property of having tests right there with the code. This was mentioned in the previous chapter as one negative effect of moving some doctests out of docstrings and into the __test__ dictionary in tests.py. The effect is worse with unit tests since all unit tests are usually kept in files separate from the code being tested. Thus there are usually no tests to be seen right near the code, which again may discourage writing tests. With unit tests, unless a methodology such as a test-driven development is employed, the "out of sight, out of mind" effect may easily result in test-writing becoming an afterthought.

Finally, we've lost the built-in documentation of the doctest version. This is more than just the potential for automatically-generated documentation from docstrings. Doctests are often more readable than unit tests, where extraneous code that is just test overhead can obscure what the test is intending to test. Note though, that using unit tests does not imply that you have to throw away doctests; it is perfectly fine to use both kinds of tests in your application. Each has their strengths, thus for many projects it is probably best to have a good mixture of unit tests and doctests rather than relying on a single type for all testing.

Revisiting the doctest caveats

In the previous chapter, we developed a list of things to watch out for when writing doctests. When discussing these, unit tests were sometimes mentioned as an alternative that did not suffer from the same problems. But are unit tests really immune to these problems, or do they just make the problems easier to avoid or address? In this section, we revisit the doctest caveats and consider how susceptible unit tests are to the same or similar issues.

Testing 1, 2, 3: Basic Unit Testing

Environmental dependence

The first doctest caveat discussed was environmental dependence: relying on the implementation details of code other than the code actually being tested. Though this type of dependence can happen with unit tests, it is less likely to occur. This is because a very common way for this type of dependence to creep into doctests is due to reliance on the printed representation of objects, as they are displayed in a Python shell session. Unit tests are far removed from the Python shell. It requires some coding effort to get an object's printed representation in a unit test, thus it is rare for this form of environmental dependence to creep into a unit test.

One common form of environmental dependence mentioned in *Chapter 2* that also afflicts unit tests involves file pathnames. Unit tests, just as doctests, need to take care that differences in file pathname conventions across operating systems do not cause bogus test failures when a test is run on an operating system different from the one where it was originally written. Thus, though unit tests are less prone to the problem of environmental dependence, they are not entirely immune.

Database dependence

Database dependence is a specific form of environmental dependence that is particularly common for Django applications to encounter. In the doctests, we saw that the initial implementation of the tests was dependent on the specifics of the message that accompanied an IntegrityError. In order to make the doctests pass on multiple different databases, we needed to modify the initial tests to ignore the details of this message.

We do not have this same problem with the unit test version. The assertRaises used to check for an expected exception already does not consider the exception message detail. For example:

```
self.assertRaises(IntegrityError, s.save)
```

There are no message specifics included there, so we don't need to do anything to ignore differences in messages from different database implementations.

In addition, unit tests make it easier to deal with even more wide-reaching differences than message details. It was noted in the previous chapter that for some configurations of MySQL, ignoring the message detail is not enough to allow all the tests to pass. The test that has a problem here is the one that ensures closes is only auto-set during initial model creation. The unit test version of this test is:

```
def testClosesReset(self):
    """Verify closes is only autoset during initial create"""
```

Chapter 3

```
s = Survey.objects.create(title=self.t, opens=self.sd)
s.closes = None
self.assertRaises(IntegrityError, s.save)
```

This test fails if it is run on a MySQL server that is running in non-strict mode. In this mode, MySQL does not raise an IntegrityError on an attempt to update a row to contain a NULL value in a column declared to be NOT NULL. Rather, the value is set to an implicit default value, and a warning is issued. Thus, we see a test error when we run this test on a MySQL server configured to run in non-strict mode:

```
______
ERROR: Verify closes is only autoset during initial create
Traceback (most recent call last):
  File "/dj_projects/marketr/survey/tests.py", line 35, in
testClosesReset
    self.assertRaises(IntegrityError, s.save)
  File "/usr/lib/python2.5/unittest.py", line 320, in failUnlessRaises
    callableObj(*args, **kwargs)
  File "/dj_projects/marketr/survey/models.py", line 38, in save
    super(Survey, self).save(**kwargs)
  File "/usr/lib/python2.5/site-packages/django/db/models/base.py", line
410, in save
    self.save base(force insert=force insert, force update=force update)
  File "/usr/lib/python2.5/site-packages/django/db/models/base.py", line
474, in save base
    rows = manager.filter(pk=pk_val)._update(values)
  File "/usr/lib/python2.5/site-packages/django/db/models/query.py", line
444, in update
    return query.execute sql(None)
  File "/usr/lib/python2.5/site-packages/django/db/models/sql/subqueries.
py", line 120, in execute_sql
   cursor = super(UpdateQuery, self).execute sql(result type)
  File "/usr/lib/python2.5/site-packages/django/db/models/sql/query.py",
line 2369, in execute_sql
    cursor.execute(sql, params)
  File "/usr/lib/python2.5/site-packages/django/db/backends/mysql/base.
py", line 84, in execute
    return self.cursor.execute(query, args)
```

Testing 1, 2, 3: Basic Unit Testing

```
File "/var/lib/python-support/python2.5/MySQLdb/cursors.py", line 168,
in execute
   if not self._defer_warnings: self._warning_check()
File "/var/lib/python-support/python2.5/MySQLdb/cursors.py", line 82,
in _warning_check
   warn(w[-1], self.Warning, 3)
File "/usr/lib/python2.5/warnings.py", line 62, in warn
   globals)
File "/usr/lib/python2.5/warnings.py", line 102, in warn_explicit
   raise message
Warning: Column 'closes' cannot be null
```

Here we see that the warning issued by MySQL causes a simple Exception to be raised, not an IntegrityError, so the test reports an error.

There is also an additional wrinkle to consider here: This behavior of raising an Exception when MySQL issues a warning is dependent on the Django DEBUG setting. MySQL warnings are turned into raised Exceptions only when DEBUG is True (as it was for the previously run test). If we set DEBUG to False in settings.py, we see yet a different form of test failure:

```
FAIL: Verify closes is only autoset during initial create

Traceback (most recent call last):

File "/dj_projects/marketr/survey/tests.py", line 35, in testClosesReset

self.assertRaises(IntegrityError, s.save)

AssertionError: IntegrityError not raised
```

In this case, MySQL allowed the save, and since DEBUG was not turned on Django did not transform the warning issued by MySQL into an Exception, so the save simply worked.

At this point, we may seriously question whether it is even worth the effort to get this test to run properly in all these different situations, given the wildly divergent observed behaviors. Perhaps we should just require that if the code is run on MySQL, the server must be configured to run in strict mode. Then the test would be fine as it is, since the previous failures would both signal a server configuration problem. However, let's assume we do need to support running on MySQL, yet we cannot impose any particular configuration requirement on MySQL, and we still need to verify whether our code is behaving properly for this test. How do we do that?

Chapter 3

Note what we are attempting to verify in this test is that our code does not auto-set closes to some value during save if it has been reset to None after initial creation. At first, it seemed that this was easily done by just checking for an IntegrityError on an attempted save. However, we've found a database configuration where we don't get an IntegrityError. Also, depending on the DEBUG setting, we may not get any error reported at all, even if our code behaves properly and leaves closes set to None during an attempted save. Can we write the test so that it reports the proper result—that is, whether our code behaves properly—in all these situations?

The answer is yes, so long as we can determine in our test code what database is in use, how it is configured, and what the DEBUG setting is. Then all we need to do is change the expected results based on the environment the test is running in. In fact, we can test for all these things with a bit of work:

```
def testClosesReset(self):
    """Verify closes is only autoset during initial create"""
    s = Survey.objects.create(title=self.t, opens=self.sd)
    s.closes = None
    strict = True
   debug = False
    from django.conf import settings
    if settings.DATABASE ENGINE == 'mysql':
        from django.db import connection
        c = connection.cursor()
        c.execute('SELECT @@SESSION.sql_mode')
        mode = c.fetchone()[0]
        if 'STRICT' not in mode:
            strict = False;
            from django.utils import importlib
            debug = importlib.import module(
                settings.SETTINGS_MODULE).DEBUG
    if strict:
        self.assertRaises(IntegrityError, s.save)
   elif debug:
        self.assertRaises(Exception, s.save)
    else:
        s.save()
        self.assertEqual(s.closes, None)
```

Testing 1, 2, 3: Basic Unit Testing

The test code starts by assuming that we are running on a database that is operating in strict mode, and set the local variable strict to True. We also assume DEBUG is False and set a local variable to reflect that. Then, if the database in use is MySQL (determined by checking the value of settings.DATABASE_ENGINE), we need to perform some further checking to see how it is configured. Consulting the MySQL documentation shows that the way to do this is to SELECT the session's sql_mode variable. If the returned value contains the string STRICT, then MySQL is operating in strict mode, otherwise it is not. We issue this query and obtain the result using Django's support for sending raw SQL to the database. If we determine that MySQL is not configured to run in strict mode, we update our local variable strict to be False.

If we get to the point where we set strict to False, that is also when the DEBUG value in settings becomes important, since it is in this case that MySQL will issue a warning instead of raising an IntegrityError for the case we are testing here. If DEBUG is True in the settings file, then warnings from MySQL will be turned into Exceptions by Django's MySQL backend. This is done by the backend using Python's warnings module. When the backend is loaded, if DEBUG is True, then a warnings filterwarnings call is issued to force all database warnings to be turned into Exceptions.

Unfortunately, at some point after the database backend is loaded and before our test code runs, the test runner will change the in-memory settings so that DEBUG is set to False. This is done so that the behavior of test code matches as closely as possible what will happen in production. However, it means that we cannot just test the value of settings. DEBUG during the test to see if DEBUG was True when the database backend was loaded. Rather, we have to re-load the settings module and check the value in the newly loaded version. We do this using the import_module function of django.utils.importlib (this is a function from Python 2.7 that was backported to be used by Django 1.1).

Finally, we know what to look for when we run our test code. If we have determined that we are running a database operating in strict mode, we assert that attempting to save our model instance with closes set to None should raise an IntegrityError. Else, if we are running in non-strict mode, but DEBUG is True in the settings file, then the attempted save should result in an Exception being raised. Otherwise the save should work, and we test the correct behavior of our code by ensuring that closes is still set to None even after the model instance has been saved.

All of that may seem like rather a lot of trouble to go through for a pretty minor test, but it illustrates how unit tests can be written to accommodate significant differences in expected behavior in different environments. Doing the same for the doctest version is not so straightforward. Thus, while unit tests clearly do not eliminate the problem of dealing with database dependence in the tests, they make it easier to write tests that account for such differences.

Chapter 3

Test interdependence

The next doctest caveat encountered in the last chapter was test interdependence. When the doctests were run on PostgreSQL, an error was encountered in the test following the first one that intentionally triggered a database error, since that error caused the database connection to enter a state where it would accept no further commands, except ones that terminated the transaction. The fix for that was to remember to "clean up" after the intentionally triggered error by including a transaction rollback after any test step that causes such an error.

Django unit tests do not suffer from this problem. The Django test case class, django.test.TestCase, ensures that the database is reset to a clean state before each test method is called. Thus, even though the testClosesReset method ends by attempting a model save that triggers an IntegrityError, no error is seen by the next test method that runs, because the database connection is reset in the interim by the django.test.TestCase code. It is not just this error situation that is cleaned up, either. Any database rows that are added, deleted, or modified by a test case method are reset to their original states before the next method is run. (Note that on most databases, the test runner can use a transaction rollback call to accomplish this very efficiently.) Thus Django unit test methods are fully isolated from any database changes that may have been performed by tests that ran before them.

Unicode

The final doctest caveat discussed in the previous chapter concerned using Unicode literals within doctests. These were observed to not work properly, due to underlying open issues in Python related to Unicode docstrings and Unicode literals within docstrings.

Unit tests do not have this problem. A straightforward unit test for the behavior of the Survey model __unicode__ method works:

Testing 1, 2, 3: Basic Unit Testing

Note that it is necessary to add the encoding declaration to the top of <code>survey/tests.py</code>, just as we did in the previous chapter for <code>survey/models.py</code>, but it is not necessary to do any manual decoding of bytestring literals to construct Unicode objects as needed to be done in the doctest version. We just need to set our variables as we normally would, create the <code>Survey</code> instance, and assert that the result of calling <code>unicode</code> on that instance produces the string we expect. Thus testing with non-ASCII data is much more straightforward when using unit tests than it is with doctests.

Providing data for unit tests

Besides not suffering from some of the disadvantages of doctests, unit tests provide some additional useful features for Django applications. One of these features is the ability to load the database with test data prior to the test run. There are a few different ways this can be done; each is discussed in detail in the following sections.

Providing data in test fixtures

The first way to provide test data for unit tests is to load them from files, called fixtures. We will cover this method by first developing an example test that can benefit from pre-loaded test data, then showing how to create a fixture file, and finally describing how to ensure that the fixture file is loaded as part of the test.

Example test that needs test data

Before jumping into the details of how to provide a test with pre-loaded data, it would help to have an example of a test that could use this feature. So far our simple tests have gotten by pretty easily by just creating the data they need as they go along. However, as we begin to test more advanced functions, we quickly run into cases were it would become burdensome for the test itself to have to create all of the data needed for a good test.

For example, consider the Question model:

```
class Question(models.Model):
    question = models.CharField(max_length=200)
    survey = models.ForeignKey(Survey)

def __unicode__(self):
    return u'%s: %s' % (self.survey, self.question)
```

α			2
Ch	lan	te.	rs

(Note that we have added a <u>__unicode__</u> method to this model. This will come in handy later in the chapter when we begin to use the admin interface to create some survey application data.)

Recall that the allowed answers for a given Question instance are stored in a separate model, Answer, which is linked to Question using a ForeignKey:

```
class Answer(models.Model):
    answer = models.CharField(max_length=200)
    question = models.ForeignKey(Question)
    votes = models.IntegerField(default=0)
```

This Answer model also tracks how many times each answer has been chosen, in its votes field. (We have not added a __unicode__ method to this model yet, since given the way we will configure admin later in the chapter, it is not yet needed.)

Now, when analyzing survey results, one of the things we will want to know about a given Question is which of its Answers was chosen most often. That is, one of the functions that a Question model will need to support is one which returns the "winning answer" for that Question. If we think about this a bit, we realize there may not be a single winning answer. There could be a tie with multiple answers getting the same number of votes. So, this winning answer method should be flexible enough to return more than one answer. Similarly, if there were no responses to the question, it would be better to return no winning answers than the whole set of allowed answers, none of which were ever chosen. Since this method (let's call it winning_answers) may return zero, one, or more results, it's probably best for consistency's sake for it to always return something like a list.

Before even starting to implement this function, then, we have a sense of the different situations it will need to handle, and what sort of test data will be useful to have in place when developing the function itself and tests for it. A good test of this routine will require at least three different questions, each with a set of answers:

- One question that has a clear winner among the answers, that is one answer
 with more votes than all of the others, so that winning_answers returns a
 single answer
- One question that has a tie among the answers, so that winning_answers returns multiple answers
- One question that gets no responses at all, so that winning_answers returns no answers

Testing 1, 2, 3: Basic Unit Testing

In addition, we should test with a Question that has no answers linked to it. This is an edge case, certainly, but we should ensure that the winning_answers function operates properly even when it seems that the data hasn't been fully set up for analysis of which answer was most popular. So, really there should be four questions in the test data, three with a set of answers and one with no answers.

Using the admin application to create test data

Creating four questions, three with several answers, in a shell session or even a program is pretty tedious, so let's use the Django admin application instead. Back in the first chapter we included django.contrib.admin in INSTALLED_APPS, so it is already loaded. Also, when we ran manage.py syncdb, the tables needed for admin were created. However, we still need to un-comment the admin-related lines in our urls.py file. When we do that urls.py should look like this:

```
# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^marketr/', include('marketr.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)),
)
```

Chapter 3

Finally, we need to provide some admin definitions for our survey application models, and register them with the admin application so that we can edit our models in the admin. Thus, we need to create a <code>survey/admin.py</code> file that looks something like this:

```
from django.contrib import admin
from survey.models import Survey, Question, Answer

class QuestionsInline(admin.TabularInline):
    model = Question
    extra = 4

class AnswersInline(admin.TabularInline):
    model = Answer

class SurveyAdmin(admin.ModelAdmin):
    inlines = [QuestionsInline]

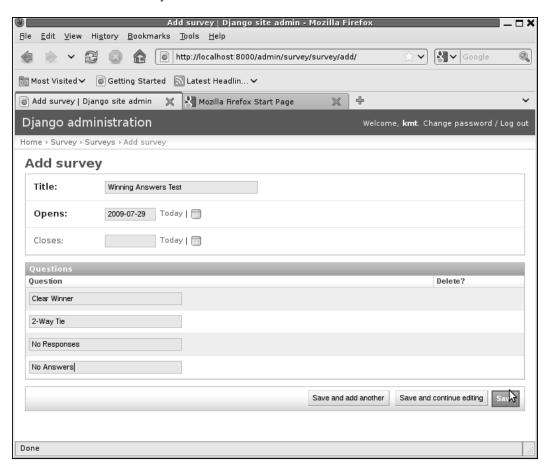
class QuestionAdmin(admin.ModelAdmin):
    inlines = [AnswersInline]

admin.site.register(Survey, SurveyAdmin)
admin.site.register(Question, QuestionAdmin)
```

Here we have mostly used the admin defaults for everything, except that we have defined and specified some admin inline classes to make it easier to edit multiple things on a single page. The way we have set up the inlines here allows us to edit Questions on the same page as the Survey they belong to, and similarly edit Answers on the same page as the Questions they are associated with. We've also specified that we want four extra empty Questions when they appear inline. The default for this value is three, but we know we want to set up four questions and we might as well set things up so we can add all four at one time.

Testing 1, 2, 3: Basic Unit Testing

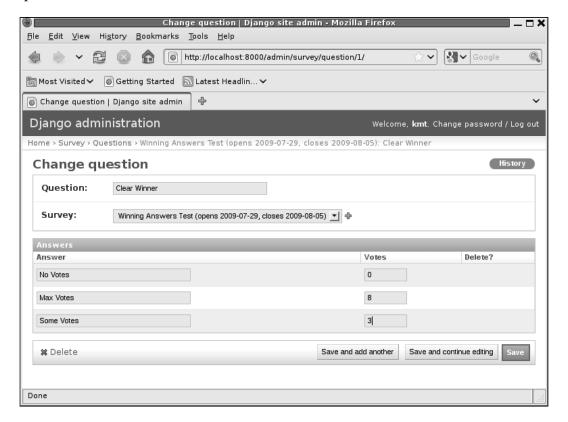
Now, we can start the development server by running python manage.py runserver in a command prompt, and access the admin application by navigating to http://localhost:8000/admin/from a browser on the same machine. After logging in as the superuser we created back in the first chapter, we'll be shown the admin main page. From there, we can click on the link to add a Survey. The **Add survey** page will let us create a survey with our four Questions:



Chapter 3

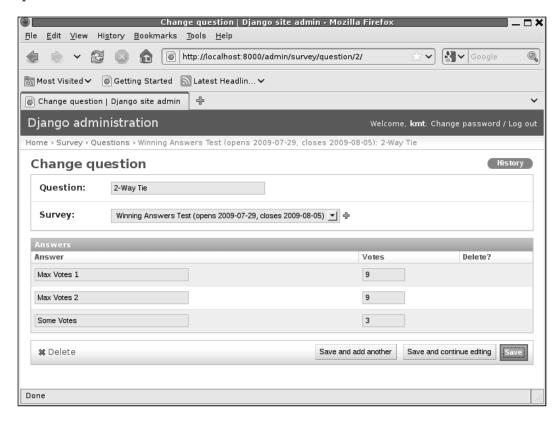
Here we've assigned our Question instances question values that are not so much questions as indications of what we are going to use each one to test. Notice this page also reflects a slight change made to the Survey model: blank=True has been added to the closes field specification. Without this change, admin would require a value to be specified here for closes. With this change, the admin application allows the field to be left blank, so that the automatic assignment done by the save override method can be used.

Once we have saved this survey, we can navigate to the change page for the first question, **Clear Winner**, and add some answers:



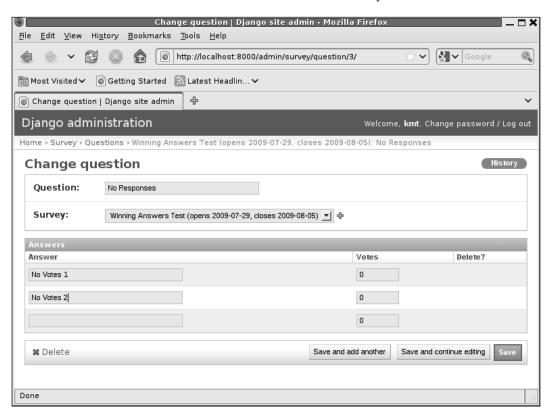
Testing 1, 2, 3: Basic Unit Testing

Thus, we set up the **Clear Winner** question to have one answer (**Max Votes**) that has more votes than all of the other answers. Similarly, we can set up the **2-Way Tie** question to have two answers that have the same number of votes:



Chapter 3

And finally, we set up the answers for **No Responses** so that we can test the situation where none of the answers to a Question have received any votes:



We do not need to do anything further with the **No Answers** question since that one is going to be used to test the case where the answer set for the question is empty, as it is when it is first created.

Writing the function itself

Now that we have our database set up with test data, we can experiment in the shell with the best way to implement the winning_answers function. As a result, we might come up with something like:

```
from django.db.models import Max

class Question(models.Model):
    question = models.CharField(max_length=200)
    survey = models.ForeignKey(Survey)
```

- [83] -

For More Information:

Testing 1, 2, 3: Basic Unit Testing

```
def winning_answers(self):
    rv = []
    max_votes = self.answer_set.aggregate(Max('votes')).values()[0]
    if max_votes and max_votes > 0:
        rv = self.answer_set.filter(votes=max_votes)
    return rv
```

The method starts by initializing a local variable rv (return value) to an empty list. Then, it uses the aggregation Max function to retrieve the maximum value for votes that exists in the set of Answer instances associated with this Question instance. That one line of code does several things in order to come up with the answer, so it may bear some more explanation. To see how it works, take a look at what each piece in turn returns in a shell session:

```
>>> from survey.models import Question
>>> q = Question.objects.get(question='Clear Winner')
>>> from django.db.models import Max
>>> q.answer_set.aggregate(Max('votes'))
{'votes max': 8}
```

Here we see that applying the aggregate function Max to the votes field of the answer_set associated with a given Question returns a dictionary containing a single key-value pair. We're only interested in the value, so we retrieve just the values from the dictionary using .values():

```
>>> q.answer_set.aggregate(Max('votes')).values()
[8]
```

However, values () returns a list and we want the single item in the list, so we retrieve it by requesting the item at index zero in the list:

```
>>> q.answer_set.aggregate(Max('votes')).values()[0]
```

Next the code tests for whether max_votes exists and if it is greater than zero (at least one answer was chosen at least once). If so, rv is reset to be the set of answers filtered down to only those that have that maximum number of votes.

Chapter 3

But when would <code>max_votes</code> not exist, since it was just set in the previous line? This can happen in the edge case where there are no answers linked to a question. In that case, the aggregate <code>Max</code> function is going to return <code>None</code> for the maximum votes value, not zero:

```
>>> q = Question.objects.get(question='No Answers')
>>> q.answer_set.aggregate(Max('votes'))
{'votes_max': None}
```

Thus in this edge case, <code>max_votes</code> may be set to <code>None</code>, so it's best to test for that and avoid trying to compare <code>None</code> to <code>0</code>. While that comparison will actually work and return what seems like a sensible answer (<code>None</code> is not greater than <code>0</code>) in Python 2.x, the attempted comparison will return a <code>TypeError</code> beginning with Python 3.0. It's wise to avoid such comparisons now so as to limit problems if and when the code needs to be ported to run under Python 3.

Finally, the function returns rv, at this point hopefully set to the correct value. (Yes, there's a bug in this function. It's more entertaining to write tests that catch bugs now and then.)

Writing a test that uses the test data

Now that we have an implementation of winning_answers, and data to test it with, we can start writing our test for the winning_answers method. We might start by adding the following test to tests.py, testing the case where there is a clear winner among the answers:

```
from survey.models import Question
class QuestionWinningAnswersTest(TestCase):
    def testClearWinner(self):
        q = Question.objects.get(question='Clear Winner')
        wa_qs = q.winning_answers()
        self.assertEqual(wa_qs.count(), 1)
        winner = wa_qs[0]
        self.assertEqual(winner.answer, 'Max Votes')
```

The test starts by retrieving the Question that has its question value set to 'Clear Winner'. Then, it calls winning_answers on that Question instance to retrieve the query set of answers for the question that received the most number of votes. Since this question is supposed to have a single winner, the test asserts that there is one element in the returned query set. It then does some further checking by retrieving the winning answer itself and verifying that its answer value is 'Max Votes'. If all that succeeds, we can be pretty sure that winning_answers returns the correct result for the case where there is a single "winner" among the answers.

For More Information:

Testing 1, 2, 3: Basic Unit Testing

Extracting the test data from the database

Now, how do we run that test against the test data we loaded via the admin application into our database? When we run the tests, they are not going to use our production database, but rather create and use an initially empty test database. This is where fixtures come in. Fixtures are just files containing data that can be loaded into the database.

The first task, then, is to extract the test data that we loaded into our production database into a fixture file. We can do this by using the manage.py dumpdata command:

python manage.py dumpdata survey --indent 4 >test_winning_answers.json

Beyond the dumpdata command itself, the various things specified there are:

- survey: This limits the dumped data to the survey application. By default, dumpdata will output data for all installed applications, but the winning answers test does not need data from any application other than survey, so we can limit the fixture file to contain only data from the survey application.
- --indent 4: This makes the data output easier to read and edit. By default, dumpdata will output the data all on a single line, which is difficult to deal with if you ever need to examine or edit the result. Specifying indent 4 makes dumpdata format the data on multiple lines, with four-space indentation making the hierarchy of structures clear. (You can specify whatever number you like for the indent value, it does not have to be 4.)
- >test_winning_answers.json: This redirects the output from the command to a file. The default output format for dumpdata is JSON, so we use .json as the file extension so that when the fixture is loaded its format will be interpreted correctly.

When dumpdata completes, we will have a test_winning_answers.json file, which contains a serialized version of our test data. Besides loading it as part of our test (which will be covered next), what might we do with this or any fixture file?

First, we can load fixtures using the manage.py loaddata command. Thus dumpdata and loaddata together provide a way to move data from one database to another. Second, we might have or write programs that process the serialized data in some way: it can sometimes be easier to perform analysis on data contained in a flat file instead of a database. Finally, the manage.py testserver command supports loading fixtures (specified on the command line) into a test database and then running the development server. This can come in handy in situations where you'd like to experiment with how a real server behaves given this test data, instead of being limited to the results of the tests written to use the data.

Chapter 3

Getting the test data loaded during the test run

Returning to our task at hand: how do we get this fixture we just created loaded when running the tests? An easy way to do this is to rename it to initial_data.json and place it in a fixtures subdirectory of our survey application directory. If we do that and run the tests, we will see that the fixture file is loaded, and our test for the clear winner case runs successfully:

```
kmt@lbox:/dj projects/marketr$ python manage.py test survey
Creating test database...
Creating table auth permission
Creating table auth_group
Creating table auth user
Creating table auth message
Creating table django_content_type
Creating table django session
Creating table django site
Creating table django admin log
Creating table survey_survey
Creating table survey_question
Creating table survey answer
Installing index for auth.Permission model
Installing index for auth.Message model
Installing index for admin.LogEntry model
Installing index for survey.Question model
Installing index for survey. Answer model
Installing json fixture 'initial_data' from '/dj_projects/marketr/survey/
fixtures'.
Installed 13 object(s) from 1 fixture(s)
Ran 9 tests in 0.079s
OK
Destroying test database...
```

Testing 1, 2, 3: Basic Unit Testing

However, that is not really the right way to get this particular fixture data loaded. Initial data fixtures are meant for constant application data that should always be there as part of the application, and this data does not fall into that category. Rather, it is specific to this particular test, and needs to be loaded only for this test. To do that, place it in the <code>survey/fixtures</code> directory with the original name, <code>test_winning_answers.json</code>. Then, update the test case code to specify that this fixture should be loaded for this test by including the file name in a <code>fixtures</code> class attribute of the test case:

```
class QuestionWinningAnswersTest(TestCase):
    fixtures = ['test_winning_answers.json']

def testClearWinner(self):
    q = Question.objects.get(question='Clear Winner')
    wa_qs = q.winning_answers()
    self.assertEqual(wa_qs.count(), 1)
    winner = wa_qs[0]
    self.assertEqual(winner.answer, 'Max Votes')
```

Note that manage.py test, at least as of Django 1.1, does not provide as much feedback for the loading of test fixtures specified this way as it does for loading initial data fixtures. In the previous test output, where the fixture was loaded as initial data, there are messages about the initial data fixture being loaded and 13 objects being installed. There are no messages like that when the fixture is loaded as part of the TestCase.

Furthermore there is no error indication if you make a mistake and specify the wrong filename in your TestCase fixtures value. For example, if you mistakenly leave the ending s off of test_winning_answers, the only indication of the problem will be that the test case fails:

```
kmt@lbox:/dj_projects/marketr$ python manage.py test survey
Creating test database...
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
```

Chapter 3

```
Creating table survey survey
Creating table survey question
Creating table survey answer
Installing index for auth.Permission model
Installing index for auth. Message model
Installing index for admin.LogEntry model
Installing index for survey.Question model
Installing index for survey. Answer model
E.....
ERROR: testClearWinner (survey.tests.QuestionWinningAnswersTest)
Traceback (most recent call last):
  File "/dj projects/marketr/survey/tests.py", line 67, in
testClearWinner
    q = Question.objects.get(question='Clear Winner')
  File "/usr/lib/python2.5/site-packages/django/db/models/manager.py",
line 120, in get
    return self.get_query_set().get(*args, **kwargs)
  File "/usr/lib/python2.5/site-packages/django/db/models/query.py", line
305, in get
    % self.model. meta.object name)
DoesNotExist: Question matching query does not exist.
Ran 9 tests in 0.066s
FAILED (errors=1)
Destroying test database...
```

Possibly the diagnostics provided for this error case may be improved in the future, but in the meantime it's best to keep in mind that mysterious errors such as that <code>DoesNotExist</code> above are likely due to the proper test fixture not being loaded rather than some error in the test code or the code being tested.

Testing 1, 2, 3: Basic Unit Testing

Now that we've got the test fixture loaded and the first test method working properly, we can add the tests for the three other cases: the one where there is a two-way tie among the answers, the one where no responses were received to a question, and the one where no answers are linked to a question. These can be written to be very similar to the existing method that tests the clear winner case:

```
def testTwoWayTie(self):
    q = Question.objects.get(question='2-Way Tie')
    wa_qs = q.winning_answers()
    self.assertEqual(wa_qs.count(), 2)
    for winner in wa_qs:
        self.assert_(winner.answer.startswith('Max Votes'))

def testNoResponses(self):
    q = Question.objects.get(question='No Responses')
    wa_qs = q.winning_answers()
    self.assertEqual(wa_qs.count(), 0)

def testNoAnswers(self):
    q = Question.objects.get(question='No Answers')
    wa_qs = q.winning_answers()
    self.assertEqual(wa_qs.count(), 0)
```

Differences are in the names of the Questions retrieved from the database, and how the specific results are tested. In the case of the 2-Way Tie, the test verifies that winning_answers returns two answers, and that both have answer values that start with 'Max Votes'. In the case of no responses, and no answers, all the tests have to do is verify that there are no items in the query set returned by winning answers.

If we now run the tests, we will find the bug that was mentioned earlier, since our last two tests fail:

```
ERROR: testNoAnswers (survey.tests.QuestionWinningAnswersTest)

Traceback (most recent call last):

File "/dj_projects/marketr/survey/tests.py", line 88, in testNoAnswers self.assertEqual(wa_qs.count(), 0)

TypeError: count() takes exactly one argument (0 given)

ERROR: testNoResponses (survey.tests.QuestionWinningAnswersTest)
```

- [90] -

For More Information:

Chapter 3

```
Traceback (most recent call last):
   File "/dj_projects/marketr/survey/tests.py", line 83, in
testNoResponses
   self.assertEqual(wa_qs.count(), 0)
TypeError: count() takes exactly one argument (0 given)
```

The problem here is that winning answers is inconsistent in what it returns:

```
def winning_answers(self):
    rv = []
    max_votes = self.answer_set.aggregate(Max('votes')).values()[0]
    if max_votes and max_votes > 0:
        rv = self.answer_set.filter(votes=max_votes)
    return rv
```

The return value rv is initialized to a list in the first line of the function, but then when it is set in the case where there are answers that received votes, it is set to be the return value from a filter call, which returns a QuerySet, not a list. The test methods, since they use count () with no arguments on the return value of winning answers, are expecting a QuerySet.

Which is more appropriate for winning_answers to return: a list or a QuerySet? Probably a QuerySet. The caller may only be interested in the count of answers in the set and not the specific answers, so it may not be necessary to retrieve the actual answers from the database. If winning_answers consistently returns a list, it would have to force the answers to be read from the database in order to put them in a list. Thus, it's probably more efficient to always return a QuerySet and let the caller's requirements dictate what ultimately needs to be read from the database. (Given the small number of items we'd expect to be in this set, there is probably little to no efficiency to be gained here, but it is still a good habit to get into in order to consider such things when designing interfaces.)

A way to fix winning_answers to always return a QuerySet is to use the none() method applied to the answer_set, which will return an empty QuerySet:

```
def winning_answers(self):
    max_votes = self.answer_set.aggregate(Max('votes')).values()[0]
    if max_votes and max_votes > 0:
        rv = self.answer_set.filter(votes=max_votes)
    else:
        rv = self.answer_set.none()
    return rv
```

Testing 1, 2, 3: Basic Unit Testing

After making this change, the complete QuestionWinningAnswersTest TestCase runs successfully.

Creating data during test set up

While test fixtures are very convenient, they are sometimes not the right tool for the job. Specifically, since the fixture files contain fixed, hard-coded values for all model data, fixtures are sometimes not flexible enough for all tests.

As an example, let's return to the Survey model and consider some methods we are likely to want it to support. Recall that a survey has both, an opens and a closes date, so at any point in time a particular Survey instance may be considered "completed", "active", or "upcoming", depending on where the current date falls in relation to the survey's opens and closes dates. It will be useful to have easy access to these different categories of surveys. The typical way to support this in Django is to create a special model Manager for Survey that implements methods to return appropriately-filtered query sets. Such a Manager might look like this:

This manager implements three methods:

- completed: This returns a QuerySet of Survey filtered down to only those with closes values earlier than today. These are surveys that are closed to any more responses.
- active: This returns a QuerySet of Survey filtered down to only those with opens values earlier or equal to today, and closes later than or equal to today. These are surveys that are open to receiving responses.
- upcoming: This returns a QuerySet of Survey filtered down to only those with opens values later than today. These are surveys that are not yet open to responses.

Chapter 3

To make this custom manager the default for the Survey model, assign an instance of it to the value of the Survey objects attribute:

```
class Survey(models.Model):
   title = models.CharField(max_length=60)
   opens = models.DateField()
   closes = models.DateField(blank=True)

   objects = SurveyManager()
```

Why might we have difficulty testing these methods using fixture data? The problem arises due to the fact that the methods rely on the moving target of today's date. It's not actually a problem for testing completed, as we can set up test data for surveys with closes dates in the past, and those closes dates will continue to be in the past no matter how much further forward in time we travel.

It is, however, a problem for active and upcoming, since eventually, even if we choose closes (and, for upcoming, opens) dates far in the future, today's date will (barring universal catastrophe) at some point catch up with those far-future dates. When that happens, the tests will start to fail. Now, we may expect that there is no way our software will still be running in that far-future time. (Or we may simply hope that we are no longer responsible for maintaining it then.) But that's not really a good approach. It would be much better to use a technique that doesn't result in time-bombs in the tests.

If we don't want to use a test fixture file with hard-coded dates to test these routines, what is the alternative? What we can do instead is much like what we were doing earlier: create the data dynamically in the test case. As noted earlier, this might be somewhat tedious, but note we do not have to re-create the data for each test method. Unit tests provide a hook method, setUp, which we can use to implement any common pre-test initialization. The test machinery will ensure that our setUp routine is run prior to each of our test methods. Thus setUp is a good place to put code that dynamically creates fixture-like data for our tests.

In a test for the custom Survey manager, then, we might have a setup routine that looks like this:

```
class SurveyManagerTest(TestCase):
    def setUp(self):
        today = datetime.date.today()
        oneday = datetime.timedelta(1)
        yesterday = today - oneday
        tomorrow = today + oneday
        Survey.objects.all().delete()
```

Testing 1, 2, 3: Basic Unit Testing

```
Survey.objects.create(title="Yesterday", opens=yesterday, closes=yesterday)

Survey.objects.create(title="Today", opens=today, closes=today)

Survey.objects.create(title="Tomorrow", opens=tomorrow, closes=tomorrow)
```

This method creates three Surveys: one that opened and closed yesterday, one that opens and closes today, and one that opens and closes tomorrow. Before it creates these, it deletes all Survey objects that are in the database. Thus, each test method in the SurveyManagerTest can rely on there being exactly three Surveys in the database, one in each of the three states.

Why does the test first delete all survey objects? There should not be any Surveys in the database yet, right? That call is there just in case at some future point, the survey application acquires an initial data fixture that includes one or more Surveys. If such a fixture existed, it would be loaded during test initialization, and would break these tests that rely on there being exactly three Surveys in the database. Thus, it is safest for setup here to ensure that the only Surveys in the database are the ones it creates.

A test for the Survey manager completed function might then be:

The test first asserts that on entry there is one completed Survey in the database. It then verifies that the one Survey returned by the completed function is in fact that actual survey it expects to be completed, that is the one with title set to "Yesterday". The test then goes a step further and modifies that completed Survey so that its closes date no longer qualifies it as completed, and saves that change to the database. When that has been done, the test asserts that there are now zero completed Surveys in the database.

Chapter 3

Testing with that routine verifies that the test works, so a similar test for active surveys might be written as:

```
def testActive(self):
    self.assertEqual(Survey.objects.active().count(), 1)
    active_survey = Survey.objects.get(title="Today")
    self.assertEqual(Survey.objects.active()[0], active_survey)
    yesterday = datetime.date.today() - datetime.timedelta(1)
    active_survey.opens = active_survey.closes = yesterday
    active_survey.save()
    self.assertEqual(Survey.objects.active().count(), 0)
```

This is very much like the test for completed. It asserts that there is one active Survey on entry, retrieves the active Survey and verifies that it is the one expected to be active, modifies it so that it no longer qualifies as active (by making it qualify as closed), saves the modification, and finally verifies that active then returns that there are no active Surveys.

Similarly, a test for upcoming surveys might be:

But won't all those tests interfere with each other? For example, the test for completed makes the "Yesterday" survey appear to be active, and the test for active makes the "Today" survey appear to be closed. It seems that whichever one runs first is going to make a change that will interfere with the correct operation of the other test.

In fact, though, the tests don't interfere with each other, because the database is reset and the test case <code>setUp</code> method is re-run before each test method is run. So <code>setUp</code> is not run once per <code>TestCase</code>, but rather once per test method within the <code>TestCase</code>. Running the tests shows that all of these tests pass, even though each updates the database in a way that would interfere with the others, if the changes it made were seen by the others:

```
testActive (survey.tests.SurveyManagerTest) ... ok
testCompleted (survey.tests.SurveyManagerTest) ... ok
testUpcoming (survey.tests.SurveyManagerTest) ... ok
```

Testing 1, 2, 3: Basic Unit Testing

There is a companion method to <code>setUp</code>, called <code>tearDown</code> that can be used to perform any cleaning up after test methods. In this case it isn't necessary, since the default Django operation of resetting the database between test method executions takes care of un-doing the database changes made by the test methods. The <code>tearDown</code> routine is useful for cleaning up any non-database changes (such as temporary file creation, for example) that may be done by the tests.

Summary

We have now covered the basics of unit testing Django applications. In this chapter, we:

- Converted the previously-written doctests for the Survey model to unit tests, which allowed us to directly compare the pros and cons of each test approach
- Revisited the doctest caveats from the previous chapter and examined to what extent (if any) unit tests are susceptible to the same issues
- Began to learn some of the additional features available with unit tests; in particular, features related to loading test data

In the next chapter, we will start investigating even more advanced features that are available to Django unit tests.

Where to buy this book

You can buy Django 1.1 Testing and Debugging from the Packt Publishing website: https://www.packtpub.com/django-1-1-testing-and-debugging/book.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



For More Information: