Django

Documentation

Testing tools

Django provides a small set of tools that come in handy when writing tests.

The test client

The test client is a Python class that acts as a dummy Web browser, allowing you to test your views and interact with your Django-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate GET and POST requests on a URL and observe the response – everything from low-level HTTP (result headers and status codes) to page content.
- See the chain of redirects (if any) and check the URL and status code at each step.
- Test that a given request is rendered by a given Django template, with a template context that contains certain values.

Note that the test client is not intended to be a replacement for Selenium or other "in-browser" frameworks. Django's test client has a different focus. In short:

- Use Django's test client to establish that the correct template is being rendered and that the template is passed the correct context data.
- Use in-browser frameworks like Selenium to test *rendered* HTML and the *behavior* of Web pages, namely JavaScript functionality. Django also provides special support for those frameworks; see the section on **`LiveServerTestCase`** for more details.

A comprehensive test suite should use a combination of both test types.

Overview and a quick example

To use the test client, instantiate **`django.test.Client`** and retrieve Web pages:

```
>>> from django.test import Client
>>> c = Client()
>>> response = c.post('/login/', {'username': 'john', 'password': 'smith'})
>>> response.status_code
200
>>> response = c.get('/customer/details/')
>>> response.content
b'<!DOCTYPE html...'
```

As this example suggests, you can instantiate **`Client`** from within a session of the Python interactive interpreter.

Note a few important things about how the test client works:

- The test client does *not* require the Web server to be running. In fact, it will run just fine with no Web server running at all! That's because it avoids the overhead of HTTP and deals directly with the Django framework. This helps make the unit tests run quickly.
- When retrieving pages, remember to specify the *path* of the URL, not the whole domain. For example, this is correct:

```
>>> c.get('/login/')
```

This is incorrect:

```
>>> c.get('https://www.example.com/login/')
```

The test client is not capable of retrieving Web pages that are not powered by your Django project. If you need to retrieve other Web pages, use a Python standard library module such as **urllib**.

- To resolve URLs, the test client uses whatever URLconf is pointed-to by your **ROOT_URLCONF** setting.
- Although the above example would work in the Python interactive interpreter, some of the test client's functionality, notably the template-related functionality, is only available *while tests are running*.

  The reason for this is that Django's test runner performs a bit of black magic in order to determine which template was loaded by a given view. This black magic (essentially a patching of Django's template system in memory) only happens during test running.
- By default, the test client will disable any CSRF checks performed by your site.

  If, for some reason, you *want* the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks. To do this, pass in the **enforce_csrf_checks** argument when you construct your client:

```
>>> from django.test import Client
>>> csrf_client = Client(enforce_csrf_checks=True)
```

Making requests

Use the **django.test.Client** class to make requests.

***class* Client(*enforce_csrf_checks=False*, ***defaults*)[source]**

It requires no arguments at time of construction. However, you can use keywords arguments to specify some default headers. For example, this will send a **User-Agent** HTTP header in each request:

```
>>> c = Client(HTTP_USER_AGENT='Mozilla/5.0')
```

The values from the **extra** keywords arguments passed to **get()**, **post()**, etc. have precedence over the defaults passed to the class constructor.

The **enforce_csrf_checks** argument can be used to test CSRF protection (see above).

Once you have a **Client** instance, you can call any of the following methods:

get(*path*, *data=None*, *follow=False*, *secure=False*, ***extra*)[source]

Makes a GET request on the provided **path** and returns a **Response** object, which is documented below.

The key-value pairs in the **data** dictionary are used to create a GET data payload. For example:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7})
```

...will result in the evaluation of a GET request equivalent to:

```
/customers/details/?name=fred&age=7
```

The **extra** keyword arguments parameter can be used to specify headers to be sent in the request. For example:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7},
...       HTTP_X_REQUESTED_WITH='XMLHttpRequest')
```

...will send the HTTP header **HTTP_X_REQUESTED_WITH** to the details view, which is a good way to test code paths that use the **django.http.HttpRequest.is_ajax()** method.

CGI specification

The headers sent via ****extra** should follow CGI specification. For example, emulating a different "Host" header as sent in the HTTP request from the browser to the server should be passed as **HTTP_HOST**.

If you already have the GET arguments in URL-encoded form, you can use that encoding instead of using the data argument. For example, the previous GET request could also be posed as:

```
>>> c = Client()
>>> c.get('/customers/details/?name=fred&age=7')
```

If you provide a URL with both an encoded GET data and a data argument, the data argument will take precedence.

If you set **follow** to **True** the client will follow any redirects and a **redirect_chain** attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you had a URL **/redirect_me/** that redirected to **/next/**, that redirected to **/final/**, this is what you'd see:

```
>>> response = c.get('/redirect_me/', follow=True)
>>> response.redirect_chain
[('http://testserver/next/', 302), ('http://testserver/final/', 302)]
```

If you set **secure** to **True** the client will emulate an HTTPS request.

post(*path*, *data=None*, *content_type=MULTIPART_CONTENT*, *follow=False*, *secure=False*, ***extra*)[source]

Makes a POST request on the provided **path** and returns a **Response** object, which is documented below.

The key-value pairs in the **data** dictionary are used to submit POST data. For example:

```
>>> c = Client()
>>> c.post('/login/', {'name': 'fred', 'passwd': 'secret'})
```

...will result in the evaluation of a POST request to this URL:

```
/login/
```

...with this POST data:

```
name=fred&passwd=secret
```

If you provide **content_type** (e.g. *text/xml* for an XML payload), the contents of **data** will be sent as-is in the POST request, using **content_type** in the HTTP **Content-Type** header.

If you don't provide a value for **content_type**, the values in **data** will be transmitted with a content type of *multipart/form-data*. In this case, the key-value pairs in **data** will be encoded as a multipart message and used to create the POST data payload.

To submit multiple values for a given key – for example, to specify the selections for a **<select multiple>** – provide the values as a list or tuple for the required key. For example, this value of **data** would submit three selected values for the field named **choices**:

```
{'choices': ('a', 'b', 'd')}
```

Submitting files is a special case. To POST a file, you need only provide the file field name as a key, and a file handle to the file you wish to upload as a value. For example:

```
>>> c = Client()
>>> with open('wishlist.doc') as fp:
...     c.post('/customers/wishes/', {'name': 'fred', 'attachment': fp})
```

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

You may also provide any file-like object (e.g., **StringIO** or **BytesIO**) as a file handle.

Note that if you wish to use the same file handle for multiple **post()** calls then you will need to manually reset the file pointer between posts. The easiest way to do this is to manually close the file after it has been provided to **post()**, as demonstrated above.

You should also ensure that the file is opened in a way that allows the data to be read. If your file contains binary data such as an image, this means you will need to open the file in **rb** (read binary) mode.

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

If the URL you request with a POST contains encoded parameters, these parameters will be made available in the request.GET data. For example, if you were to make the request:

```
>>> c.post('/login/?visitor=true', {'name': 'fred', 'passwd': 'secret'})
```

... the view handling this request could interrogate request.POST to retrieve the username and password, and could interrogate request.GET to determine if the user was a visitor.

If you set **follow** to **True** the client will follow any redirects and a **redirect_chain** attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you set **secure** to **True** the client will emulate an HTTPS request.

head(*path*, *data=None*, *follow=False*, *secure=False*, ***extra*)**[source]**

Makes a HEAD request on the provided **path** and returns a **Response** object. This method works just like **Client.get()**, including the **follow**, **secure** and **extra** arguments, except it does not return a message body.

options(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)**[source]**

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

When **data** is provided, it is used as the request body, and a **Content-Type** header is set to **content_type**.

The **follow**, **secure** and **extra** arguments act the same as for **Client.get()**.

put(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)**[source]**

Makes a PUT request on the provided **path** and returns a **Response** object. Useful for testing RESTful interfaces.

When **data** is provided, it is used as the request body, and a **Content-Type** header is set to **content_type**.

The **follow**, **secure** and **extra** arguments act the same as for **Client.get()**.

patch(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)**[source]**

Makes a PATCH request on the provided **path** and returns a **Response** object. Useful for testing RESTful interfaces.

The **follow**, **secure** and **extra** arguments act the same as for **Client.get()**.

delete(*path*, *data=''*, *content_type='application/octet-stream'*, *follow=False*, *secure=False*, ***extra*)**[source]**

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

When **data** is provided, it is used as the request body, and a **Content-Type** header is set to **content_type**.

The **follow**, **secure** and **extra** arguments act the same as for **Client.get()**.

trace(*path*, *follow=False*, *secure=False*, ***extra*)**[source]**

Makes a TRACE request on the provided **path** and returns a **Response** object. Useful for simulating diagnostic probes.

Unlike the other request methods, **data** is not provided as a keyword parameter in order to comply with **RFC 7231#section-4.3.8**, which mandates that TRACE requests must not have a body.

The **follow**, **secure**, and **extra** arguments act the same as for **Client.get()**.

login(***credentials*)[source]

If your site uses Django's authentication system and you deal with logging in users, you can use the test client's **login()** method to simulate the effect of a user logging into the site.

After you call this method, the test client will have all the cookies and session data required to pass any login-based tests that may form part of a view.

The format of the **credentials** argument depends on which authentication backend you're using (which is configured by your **AUTHENTICATION_BACKENDS** setting). If you're using the standard authentication backend provided by Django (**ModelBackend**), **credentials** should be the user's username and password, provided as keyword arguments:

```
>>> c = Client()
>>> c.login(username='fred', password='secret')

# Now you can access a view that's only available to logged-in users.
```

If you're using a different authentication backend, this method may require different credentials. It requires whichever credentials are required by your backend's **authenticate()** method.

login() returns **True** if it the credentials were accepted and login was successful.

Finally, you'll need to remember to create user accounts before you can use this method. As we explained above, the test runner is executed using a test database, which contains no users by default. As a result, user accounts that are valid on your production site will not work under test conditions. You'll need to create users as part of the test suite – either manually (using the Django model API) or with a test fixture. Remember that if you want your test user to have a password, you can't set the user's password by setting the password attribute directly – you must use the **set_password()** function to store a correctly hashed password. Alternatively, you can use the **create_user()** helper method to create a new user with a correctly hashed password.

Changed in Django 1.10:
In previous versions, inactive users (**is_active=False**) were not permitted to login.

force_login(*user*, *backend=None*)[source]

New in Django 1.9.

If your site uses Django's authentication system, you can use the **force_login()** method to simulate the effect of a user logging into the site. Use this method instead of **login()** when a test requires a user be logged in and the details of how a user logged in aren't important.

Unlike **login()**, this method skips the authentication and verification steps: inactive users (**is_active=False**) are permitted to login and the user's credentials don't need to be provided.

The user will have its **backend** attribute set to the value of the **backend** argument (which should be a dotted Python path string), or to **settings.AUTHENTICATION_BACKENDS[0]** if a value isn't provided. The **authenticate()** function called by **login()** normally annotates the user like this.

This method is faster than **login()** since the expensive password hashing algorithms are bypassed. Also, you can speed up **login()** by using a weaker hasher while testing.

logout()[source]

If your site uses Django's authentication system, the **logout()** method can be used to simulate the effect of a user logging out of your site.

After you call this method, the test client will have all the cookies and session data cleared to defaults. Subsequent requests will appear to come from an **AnonymousUser**.

Testing responses

The **get()** and **post()** methods both return a **Response** object. This **Response** object is *not* the same as the **HttpResponse** object returned by Django views; the test response object has some additional data useful for test code to verify.

Specifically, a **Response** object has the following attributes:

class Response

client

The test client that was used to make the request that resulted in the response.

content

The body of the response, as a bytestring. This is the final page content as rendered by the view, or any error message.

context

The template **Context** instance that was used to render the template that produced the response content.

If the rendered page used multiple templates, then **context** will be a list of **Context** objects, in the order in which they were rendered.

Regardless of the number of templates used during rendering, you can retrieve context values using the **[]** operator. For example, the context variable **name** could be retrieved using:

```
>>> response = client.get('/foo/')
>>> response.context['name']
'Arthur'
```

Not using Django templates?

This attribute is only populated when using the **DjangoTemplates** backend. If you're using another template engine, **context_data** may be a suitable alternative on responses with that attribute.

json(*****kwargs***)

New in Django 1.9.

The body of the response, parsed as JSON. Extra keyword arguments are passed to **json.loads()**. For example:

```
>>> response = client.get('/foo/')
>>> response.json()['name']
'Arthur'
```

If the **Content-Type** header is not **"application/json"**, then a **ValueError** will be raised when trying to parse the response.

request

The request data that stimulated the response.

wsgi_request

The **WSGIRequest** instance generated by the test handler that generated the response.

status_code

The HTTP status of the response, as an integer. For a full list of defined codes, see the IANA status code registry.

templates

A list of **Template** instances used to render the final content, in the order they were rendered. For each template in the list, use **template.name** to get the template's file name, if the template was loaded from a file. (The name is a string such as **'admin/index.html'**.)

Not using Django templates?

This attribute is only populated when using the **DjangoTemplates** backend. If you're using another template engine, **template_name** may be a suitable alternative if you only need the name of the template used for rendering.

resolver_match

An instance of **ResolverMatch** for the response. You can use the **func** attribute, for example, to verify the view that served the response:

```
# my_view here is a function based view
self.assertEqual(response.resolver_match.func, my_view)

# class-based views need to be compared by name, as the functions
# generated by as_view() won't be equal
self.assertEqual(response.resolver_match.func.__name__, MyView.as_view().__name__)
```

If the given URL is not found, accessing this attribute will raise a **Resolver404** exception.

You can also use dictionary syntax on the response object to query the value of any settings in the HTTP headers. For example, you could determine the content type of a response using **response['Content-Type']**.

Exceptions

If you point the test client at a view that raises an exception, that exception will be visible in the test case. You can then use a standard **try ... except** block or **assertRaises()** to test for exceptions.

The only exceptions that are not visible to the test client are **Http404**, **PermissionDenied**, **SystemExit**, and **SuspiciousOperation**. Django catches these exceptions internally and converts them into the appropriate HTTP response codes. In these cases, you can check **response.status_code** in your test.

Persistent state

The test client is stateful. If a response returns a cookie, then that cookie will be stored in the test client and sent with all subsequent **get()** and **post()** requests.

Expiration policies for these cookies are not followed. If you want a cookie to expire, either delete it manually or create a new **Client** instance (which will effectively delete all cookies).

A test client has two attributes that store persistent state information. You can access these properties as part of a test condition.

Client.cookies

A Python **SimpleCookie** object, containing the current values of all the client cookies. See the documentation of the **http.cookies** module for more.

Client.session

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

To modify the session and then save it, it must be stored in a variable first (because a new **SessionStore** is created every time this property is accessed):

```
def test_something(self):
    session = self.client.session
    session['somekey'] = 'test'
    session.save()
```

Setting the language

When testing applications that support internationalization and localization, you might want to set the language for a test client request. The method for doing so depends on whether or not the **LocaleMiddleware** is enabled.

If the middleware is enabled, the language can be set by creating a cookie with a name of **LANGUAGE_COOKIE_NAME** and a value of the language code:

```
from django.conf import settings

def test_language_using_cookie(self):
    self.client.cookies.load({settings.LANGUAGE_COOKIE_NAME: 'fr'})
    response = self.client.get('/')
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

or by including the **Accept-Language** HTTP header in the request:

```
def test_language_using_header(self):
    response = self.client.get('/', HTTP_ACCEPT_LANGUAGE='fr')
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

More details are in How Django discovers language preference.

If the middleware isn't enabled, the active language may be set using **`translation.override()`**:

```
from django.utils import translation

def test_language_using_override(self):
    with translation.override('fr'):
        response = self.client.get('/')
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

More details are in Explicitly setting the active language.

Example

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

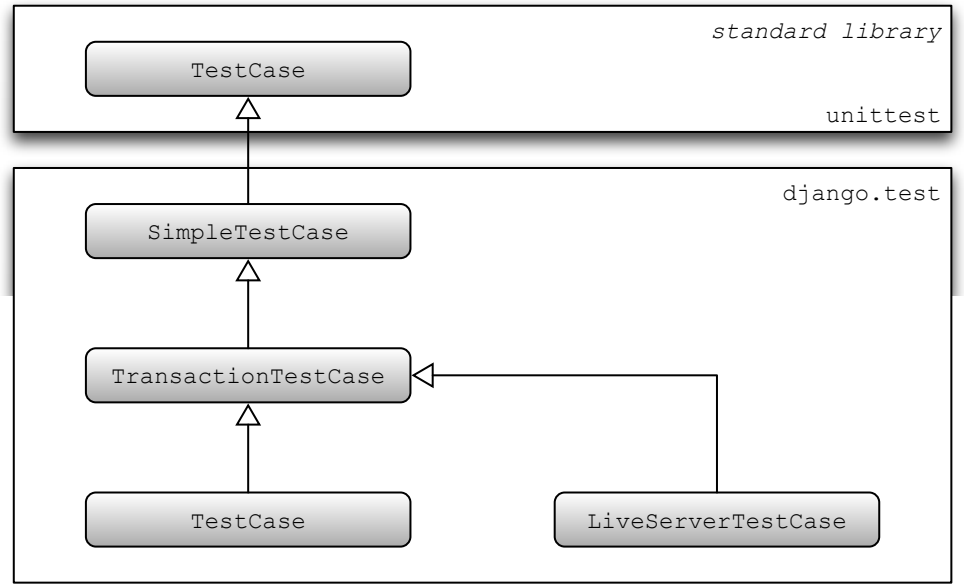
```
import unittest
from django.test import Client

class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Every test needs a client.
        self.client = Client()

    def test_details(self):
        # Issue a GET request.
        response = self.client.get('/customer/details/')

        # Check that the response is 200 OK.
        self.assertEqual(response.status_code, 200)

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

See also

`django.test.RequestFactory`

Provided test case classes

Normal Python unit test classes extend a base class of **`unittest.TestCase`**. Django provides a few extensions of this base class:



Hierarchy of Django unit testing classes

Converting a normal **`unittest.TestCase`** to any of the subclasses is easy: change the base class of your test from **`unittest.TestCase`** to the subclass. All of the standard Python unit test functionality will be available, and it will be augmented with some useful additions as described in each section below.

SimpleTestCase

class **`SimpleTestCase`**[source]

A subclass of **`unittest.TestCase`** that adds this functionality:

- Some useful assertions like:
  - Checking that a callable **`raises a certain exception`**.
  - Testing form field **`rendering and error treatment`**.
  - Testing **`HTML responses for the presence/lack of a given fragment`**.
  - Verifying that a template **`has/hasn't been used to generate a given response content`**.
  - Verifying a HTTP **`redirect`** is performed by the app.
  - Robustly testing two **`HTML fragments`** for equality/inequality or **`containment`**.
  - Robustly testing two **`XML fragments`** for equality/inequality.
  - Robustly testing two **`JSON fragments`** for equality.
- The ability to run tests with modified settings.
- Using the **`client Client`**.

If your tests make any database queries, use subclasses **`TransactionTestCase`** or **`TestCase`**.

`SimpleTestCase.allow_database_queries`

> **New in Django 1.9.**

`SimpleTestCase` disallows database queries by default. This helps to avoid executing write queries which will affect other tests since each **`SimpleTestCase`** test isn't run in a transaction. If you aren't concerned about this problem, you can disable this behavior by setting the **`allow_database_queries`** class attribute to **`True`** on your test class.

> **Warning**
>
> **`SimpleTestCase`** and its subclasses (e.g. **`TestCase`**, ...) rely on **`setUpClass()`** and **`tearDownClass()`** to perform some class-wide initialization (e.g. overriding settings). If you need to override those methods, don't forget to call the **`super`** implementation:

```
class MyTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super(MyTestCase, cls).setUpClass()
        ...

    @classmethod
    def tearDownClass(cls):
        ...
        super(MyTestCase, cls).tearDownClass()
```

Be sure to account for Python's behavior if an exception is raised during **`setUpClass()`**. If that happens, neither the tests in the class nor **`tearDownClass()`** are run. In the case of **`django.test.TestCase`**, this will leak the transaction created in **`super()`** which results in various symptoms including a segmentation fault on some platforms (reported on OS X). If you want to intentionally raise an exception such as **`unittest.SkipTest`** in **`setUpClass()`**, be sure to do it before calling **`super()`** to avoid this.

`TransactionTestCase`

class **`TransactionTestCase`**[source]

`TransactionTestCase` inherits from **`SimpleTestCase`** to add some database-specific features:

- Resetting the database to a known state at the beginning of each test to ease testing and using the ORM.
- Database **`fixtures`**.
- Test skipping based on database backend features.
- The remaining specialized **`assert*`** methods.

Django's **`TestCase`** class is a more commonly used subclass of **`TransactionTestCase`** that makes use of database transaction facilities to speed up the process of resetting the database to a known state at the beginning of each test. A consequence of this, however, is that some database behaviors cannot be tested within a Django **`TestCase`** class. For instance, you cannot test that a block of code is executing within a transaction, as is required when using **`select_for_update()`**. In those cases, you should use **`TransactionTestCase`**.

`TransactionTestCase` and **`TestCase`** are identical except for the manner in which the database is reset to a known state and the ability for test code to test the effects of commit and rollback:

- A **`TransactionTestCase`** resets the database after the test runs by truncating all tables. A **`TransactionTestCase`** may call commit and rollback and observe the effects of these calls on the database.
- A **`TestCase`**, on the other hand, does not truncate tables after a test. Instead, it encloses the test code in a database transaction that is rolled back at the end of the test. This guarantees that the rollback at the end of the test restores the database to its initial state.

Warning

`TestCase` running on a database that does not support rollback (e.g. MySQL with the MyISAM storage engine), and all instances of **`TransactionTestCase`**, will roll back at the end of the test by deleting all data from the test database.

Apps will not see their data reloaded; if you need this functionality (for example, third-party apps should enable this) you can set **`serialized_rollback = True`** inside the **`TestCase`** body.

`TestCase`

class **`TestCase`**[source]

This is the most common class to use for writing tests in Django. It inherits from **`TransactionTestCase`** (and by extension **`SimpleTestCase`**). If your Django application doesn't use a database, use **`SimpleTestCase`**.

The class:

- Wraps the tests within two nested **`atomic()`** blocks: one for the whole class and one for each test. Therefore, if you want to test some specific database transaction behavior, use **`TransactionTestCase`**.
- Checks deferrable database constraints at the end of each test.

Changed in Django 1.10:
The check for deferrable database constraints at the end of each test was added.

It also provides an additional method:

classmethod `TestCase.setUpTestData`()[source]

The class-level **`atomic`** block described above allows the creation of initial data at the class level, once for the whole **`TestCase`**. This technique allows for faster tests as compared to using **`setUp()`**.

For example:

```
from django.test import TestCase

class MyTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Set up data for the whole TestCase
        cls.foo = Foo.objects.create(bar="Test")
        ...

    def test1(self):
        # Some test using self.foo
        ...

    def test2(self):
        # Some other test using self.foo
        ...
```

Note that if the tests are run on a database with no transaction support (for instance, MySQL with the MyISAM engine), **`setUpTestData()`** will be called before each test, negating the speed benefits.

Be careful not to modify any objects created in **`setUpTestData()`** in your test methods. Modifications to in-memory objects from setup work done at the class level will persist between test methods. If you do need to modify them, you could reload them in the **`setUp()`** method with **`refresh_from_db()`**, for example.

`LiveServerTestCase`

***class* `LiveServerTestCase`[source]**

`LiveServerTestCase` does basically the same as **`TransactionTestCase`** with one extra feature: it launches a live Django server in the background on setup, and shuts it down on teardown. This allows the use of automated test clients other than the Django dummy client such as, for example, the Selenium client, to execute a series of functional tests inside a browser and simulate a real user's actions.

By default the live server listens on **`localhost`** and picks the first available port in the **`8081-8179`** range. Its full URL can be accessed with **`self.live_server_url`** during the tests.

Changed in Django 1.9:
In earlier versions, the live server's default address was always **`'localhost:8081'`**.

If you'd like to select another address, you may pass a different one using the **`test --liveserver`** option, for example:

```
$ ./manage.py test --liveserver=localhost:8082
```

Changed in Django 1.9:
In older versions **`live_server_url`** could only be accessed from an instance. It now is a class property and can be accessed from class methods like **`setUpClass()`**.

Another way of changing the default server address is by setting the *DJANGO_LIVE_TEST_SERVER_ADDRESS* environment variable somewhere in your code (for example, in a custom test runner):

```
import os
os.environ['DJANGO_LIVE_TEST_SERVER_ADDRESS'] = 'localhost:8082'
```

In the case where the tests are run by multiple processes in parallel (for example, in the context of several simultaneous continuous integration builds), the processes will compete for the same address, and therefore your tests might randomly fail with an “Address already in use” error. To avoid this problem, you can pass a comma-separated list of ports or ranges of ports (at least as many as the number of potential parallel processes). For example:

```
$ ./manage.py test --liveserver=localhost:8082,8090-8100,9000-9200,7041
```

Then, during test execution, each new live test server will try every specified port until it finds one that is free and takes it.

To demonstrate how to use **`LiveServerTestCase`**, let’s write a simple Selenium test. First of all, you need to install the selenium package into your Python path:

```
$ pip install selenium
```

Then, add a **`LiveServerTestCase`**-based test to your app’s tests module (for example: **`myapp/tests.py`**). For this example, we’ll assume you’re using the **`staticfiles`** app and want to have static files served during the execution of your tests similar to what we get at development time with **`DEBUG=True`**, i.e. without having to collect them using **`collectstatic`**. We’ll use the **`StaticLiveServerTestCase`** subclass which provides that functionality. Replace it with **`django.test.LiveServerTestCase`** if you don’t need that.

The code for this test may look as follows:

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium.webdriver.firefox.webdriver import WebDriver

class MySeleniumTests(StaticLiveServerTestCase):
    fixtures = ['user-data.json']

    @classmethod
    def setUpClass(cls):
        super(MySeleniumTests, cls).setUpClass()
        cls.selenium = WebDriver()
        cls.selenium.implicitly_wait(10)

    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super(MySeleniumTests, cls).tearDownClass()

    def test_login(self):
        self.selenium.get('%s%s' % (self.live_server_url, '/login/'))
        username_input = self.selenium.find_element_by_name("username")
        username_input.send_keys('myuser')
        password_input = self.selenium.find_element_by_name("password")
        password_input.send_keys('secret')
        self.selenium.find_element_by_xpath('//input[@value="Log in"]').click()
```

Finally, you may run the test as follows:

```
$ ./manage.py test myapp.tests.MySeleniumTests.test_login
```

This example will automatically open Firefox then go to the login page, enter the credentials and press the “Log in” button. Selenium offers other drivers in case you do not have Firefox installed or wish to use another browser. The example above is just a tiny fraction of what the Selenium client can do; check out the full reference for more details.

Note

When using an in-memory SQLite database to run the tests, the same database connection will be shared by two threads in parallel: the thread in which the live server is run and the thread in which the test case is run. It's important to prevent simultaneous database queries via this shared connection by the two threads, as that may sometimes randomly cause the tests to fail. So you need to ensure that the two threads don't access the database at the same time. In particular, this means that in some cases (for example, just after clicking a link or submitting a form), you might need to check that a response is received by Selenium and that the next page is loaded before proceeding with further test execution. Do this, for example, by making Selenium wait until the **<body>** HTML tag is found in the response (requires Selenium > 2.13):

```
def test_login(self):
    from selenium.webdriver.support.wait import WebDriverWait
    timeout = 2
    ...
    self.selenium.find_element_by_xpath('//input[@value="Log in"]').click()
    # Wait until the response is received
    WebDriverWait(self.selenium, timeout).until(
        lambda driver: driver.find_element_by_tag_name('body'))
```

The tricky thing here is that there's really no such thing as a "page load," especially in modern Web apps that generate HTML dynamically after the server generates the initial document. So, simply checking for the presence of **<body>** in the response might not necessarily be appropriate for all use cases. Please refer to the Selenium FAQ and Selenium documentation for more information.

Test cases features

Default test client

SimpleTestCase.client

Every test case in a **django.test.*TestCase** instance has access to an instance of a Django test client. This client can be accessed as **self.client**. This client is recreated for each test, so you don't have to worry about state (such as cookies) carrying over from one test to another.

This means, instead of instantiating a **Client** in each test:

```
import unittest
from django.test import Client

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

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

...you can just refer to **self.client**, like so:

```
from django.test import TestCase

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

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

Customizing the test client

`SimpleTestCase.client_class`

If you want to use a different **`Client`** class (for example, a subclass with customized behavior), use the **`client_class`** class attribute:

```
from django.test import TestCase, Client

class MyTestClient(Client):
    # Specialized methods for your environment
    ...

class MyTest(TestCase):
    client_class = MyTestClient

    def test_my_stuff(self):
        # Here self.client is an instance of MyTestClient...
        call_some_test_code()
```

Fixture loading

`TransactionTestCase.fixtures`

A test case for a database-backed website isn't much use if there isn't any data in the database. Tests are more readable and it's more maintainable to create objects using the ORM, for example in **`TestCase.setUpTestData()`**, however, you can also use fixtures.

A fixture is a collection of data that Django knows how to import into a database. For example, if your site has user accounts, you might set up a fixture of fake user accounts in order to populate your database during tests.

The most straightforward way of creating a fixture is to use the **`manage.py dumpdata`** command. This assumes you already have some data in your database. See the **`dumpdata` documentation** for more details.

Once you've created a fixture and placed it in a **`fixtures`** directory in one of your **`INSTALLED_APPS`**, you can use it in your unit tests by specifying a **`fixtures`** class attribute on your **`django.test.TestCase`** subclass:

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    fixtures = ['mammals.json', 'birds']

    def setUp(self):
        # Test definitions as before.
        call_setup_methods()

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

Here's specifically what will happen:

- At the start of each test, before **`setUp()`** is run, Django will flush the database, returning the database to the state it was in directly after **`migrate`** was called.
- Then, all the named fixtures are installed. In this example, Django will install any JSON fixture named **`mammals`**, followed by any fixture named **`birds`**. See the **`loaddata`** documentation for more details on defining and installing fixtures.

For performance reasons, **`TestCase`** loads fixtures once for the entire test class, before **`setUpTestData()`**, instead of before each test, and it uses transactions to clean the database before each test. In any case, you can be certain that the outcome of a test will not be affected by another test or by the order of test execution.

By default, fixtures are only loaded into the **`default`** database. If you are using multiple databases and set **`multi_db=True`**, fixtures will be loaded into all databases.

URLconf configuration

If your application provides views, you may want to include tests that use the test client to exercise those views. However, an end user is free to deploy the views in your application at any URL of their choosing. This means that your tests can't rely upon the fact that your views will be available at a particular URL. Decorate your test class or test method with **`@override_settings(ROOT_URLCONF=...)`** for URLconf configuration.

Multi-database support

`TransactionTestCase.multi_db`

Django sets up a test database corresponding to every database that is defined in the **DATABASES** definition in your settings file. However, a big part of the time taken to run a Django TestCase is consumed by the call to **`flush`** that ensures that you have a clean database at the start of each test run. If you have multiple databases, multiple flushes are required (one for each database), which can be a time consuming activity – especially if your tests don't need to test multi-database activity.

As an optimization, Django only flushes the **`default`** database at the start of each test run. If your setup contains multiple databases, and you have a test that requires every database to be clean, you can use the **`multi_db`** attribute on the test suite to request a full flush.

For example:

```
class TestMyViews(TestCase):
    multi_db = True

    def test_index_page_view(self):
        call_some_test_code()
```

This test case will flush *all* the test databases before running **`test_index_page_view`**.

The **`multi_db`** flag also affects into which databases the attr:*TransactionTestCase.fixtures* are loaded. By default (when **`multi_db=False`**), fixtures are only loaded into the **`default`** database. If **`multi_db=True`**, fixtures are loaded into all databases.

Overriding settings

Warning

Use the functions below to temporarily alter the value of settings in tests. Don't manipulate **`django.conf.settings`** directly as Django won't restore the original values after such manipulations.

`SimpleTestCase.settings`()[source]

For testing purposes it's often useful to change a setting temporarily and revert to the original value after running the testing code. For this use case Django provides a standard Python context manager (see **PEP 343**) called **`settings()`**, which can be used like this:

```
from django.test import TestCase

class LoginTestCase(TestCase):

    def test_login(self):

        # First check for the default behavior
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/accounts/login/?next=/sekrit/')

        # Then override the LOGIN_URL setting
        with self.settings(LOGIN_URL='/other/login/'):
            response = self.client.get('/sekrit/')
            self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

This example will override the **`LOGIN_URL`** setting for the code in the **`with`** block and reset its value to the previous state afterwards.

SimpleTestCase.modify_settings()[source]

It can prove unwieldy to redefine settings that contain a list of values. In practice, adding or removing values is often sufficient. The **modify_settings()** context manager makes it easy:

```
from django.test import TestCase

class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        with self.modify_settings(MIDDLEWARE={
            'append': 'django.middleware.cache.FetchFromCacheMiddleware',
            'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
            'remove': [
                'django.contrib.sessions.middleware.SessionMiddleware',
                'django.contrib.auth.middleware.AuthenticationMiddleware',
                'django.contrib.messages.middleware.MessageMiddleware',
            ],
        }):
            response = self.client.get('/')
            # ...
```

For each action, you can supply either a list of values or a string. When the value already exists in the list, **append** and **prepend** have no effect; neither does **remove** when the value doesn't exist.

override_settings()[source]

In case you want to override a setting for a test method, Django provides the **override_settings()** decorator (see **PEP 318**). It's used like this:

```
from django.test import TestCase, override_settings

class LoginTestCase(TestCase):

    @override_settings(LOGIN_URL='/other/login/')
    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

The decorator can also be applied to **TestCase** classes:

```
from django.test import TestCase, override_settings

@override_settings(LOGIN_URL='/other/login/')
class LoginTestCase(TestCase):

    def test_login(self):
        response = self.client.get('/sekrit/')
        self.assertRedirects(response, '/other/login/?next=/sekrit/')
```

modify_settings()[source]

Likewise, Django provides the **modify_settings()** decorator:

Overridden settings	Data reset

```
from django.test import TestCase, modify_settings

class MiddlewareTestCase(TestCase):

    @modify_settings(MIDDLEWARE={
        'append': 'django.middleware.cache.FetchFromCacheMiddleware',
        'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
    })
    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

The decorator can also be applied to test case classes:

```
from django.test import TestCase, modify_settings

@modify_settings(MIDDLEWARE={
    'append': 'django.middleware.cache.FetchFromCacheMiddleware',
    'prepend': 'django.middleware.cache.UpdateCacheMiddleware',
})
class MiddlewareTestCase(TestCase):

    def test_cache_middleware(self):
        response = self.client.get('/')
        # ...
```

Note

When given a class, these decorators modify the class directly and return it; they don't create and return a modified copy of it. So if you try to tweak the above examples to assign the return value to a different name than **`LoginTestCase`** or **`MiddlewareTestCase`**, you may be surprised to find that the original test case classes are still equally affected by the decorator. For a given class, **`modify_settings()`** is always applied after **`override_settings()`**.

Warning

The settings file contains some settings that are only consulted during initialization of Django internals. If you change them with **`override_settings`**, the setting is changed if you access it via the **`django.conf.settings`** module, however, Django's internals access it differently. Effectively, using **`override_settings()`** or **`modify_settings()`** with these settings is probably not going to do what you expect it to do.

We do not recommend altering the **`DATABASES`** setting. Altering the **`CACHES`** setting is possible, but a bit tricky if you are using internals that make using of caching, like **`django.contrib.sessions`**. For example, you will have to reinitialize the session backend in a test that uses cached sessions and overrides **`CACHES`**.

Finally, avoid aliasing your settings as module-level constants as **`override_settings()`** won't work on such values since they are only evaluated the first time the module is imported.

You can also simulate the absence of a setting by deleting it after settings have been overridden, like this:

```
@override_settings()
def test_something(self):
    del settings.LOGIN_URL
    ...
```

When overriding settings, make sure to handle the cases in which your app's code uses a cache or similar feature that retains state even if the setting is changed. Django provides the **`django.test.signals.setting_changed`** signal that lets you register callbacks to clean up and otherwise reset state when settings are changed.

Django itself uses this signal to reset various data:

Overridden settings	Data reset
USE_TZ, TIME_ZONE	Databases timezone
TEMPLATES	Template engines

Overridden settings	Data reset
SERIALIZATION_MODULES	Serializers cache
LOCALE_PATHS, LANGUAGE_CODE	Default translation and loaded translations
MEDIA_ROOT, DEFAULT_FILE_STORAGE	Default file storage

Emptying the test outbox

If you use any of Django's custom **`TestCase`** classes, the test runner will clear the contents of the test email outbox at the start of each test case.

For more detail on email services during tests, see Email services below.

Assertions

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

The failure messages given by most of these assertion methods can be customized with the **`msg_prefix`** argument. This string will be prefixed to any failure message generated by the assertion. This allows you to provide additional details that may help you to identify the location and cause of a failure in your test suite.

`SimpleTestCase.assertRaisesMessage`(*expected_exception*, *expected_message*, *callable*, **args*, ***kwargs*)[source]
`SimpleTestCase.assertRaisesMessage`(*expected_exception*, *expected_message*)

Asserts that execution of **`callable`** raises **`expected_exception`** and that **`expected_message`** is found in the exception's message. Any other outcome is reported as a failure. It's a simpler version of **`unittest.TestCase.assertRaisesRegex()`** with the difference that **`expected_message`** isn't treated as a regular expression.

If only the **`expected_exception`** and **`expected_message`** parameters are given, returns a context manager so that the code being tested can be written inline rather than as a function:

```
with self.assertRaisesMessage(ValueError, 'invalid literal for int()'):
    int('a')
```

Deprecated since version 1.9:
Passing **`callable`** as a keyword argument called **`callable_obj`** is deprecated. Pass the callable as a positional argument instead.

`SimpleTestCase.assertFieldOutput`(*fieldclass*, *valid*, *invalid*, *field_args=None*, *field_kwargs=None*, *empty_value=''*)[source]

Asserts that a form field behaves correctly with various inputs.

Parameters:

- **fieldclass** – the class of the field to be tested.
- **valid** – a dictionary mapping valid inputs to their expected cleaned values.
- **invalid** – a dictionary mapping invalid inputs to one or more raised error messages.
- **field_args** – the args passed to instantiate the field.
- **field_kwargs** – the kwargs passed to instantiate the field.
- **empty_value** – the expected clean output for inputs in **`empty_values`**.

For example, the following code tests that an **`EmailField`** accepts **`a@a.com`** as a valid email address, but rejects **`aaa`** with a reasonable error message:

```
self.assertFieldOutput(EmailField, {'a@a.com': 'a@a.com'}, {'aaa': ['Enter a valid email address.']})
```

`SimpleTestCase.assertFormError`(*response*, *form*, *field*, *errors*, *msg_prefix=''*)[source]

Asserts that a field on a form raises the provided list of errors when rendered on the form.

`form` is the name the **`Form`** instance was given in the template context.

`field` is the name of the field on the form to check. If **`field`** has a value of **`None`**, non-field errors (errors you can access via **`form.non_field_errors()`**) will be checked.

`errors` is an error string, or a list of error strings, that are expected as a result of form validation.

SimpleTestCase.assertFormsetError(*response*, *formset*, *form_index*, *field*, *errors*, *msg_prefix=''*)[source]

Asserts that the **formset** raises the provided list of errors when rendered.

formset is the name the **Formset** instance was given in the template context.

form_index is the number of the form within the **Formset**. If **form_index** has a value of **None**, non-form errors (errors you can access via **formset.non_form_errors()**) will be checked.

field is the name of the field on the form to check. If **field** has a value of **None**, non-field errors (errors you can access via **form.non_field_errors()**) will be checked.

errors is an error string, or a list of error strings, that are expected as a result of form validation.

SimpleTestCase.assertContains(*response*, *text*, *count=None*, *status_code=200*, *msg_prefix=''*, *html=False*)[source]

Asserts that a **Response** instance produced the given **status_code** and that **text** appears in the content of the response. If **count** is provided, **text** must occur exactly **count** times in the response.

Set **html** to **True** to handle **text** as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See **assertHTMLEqual()** for more details.

SimpleTestCase.assertNotContains(*response*, *text*, *status_code=200*, *msg_prefix=''*, *html=False*)[source]

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

Set **html** to **True** to handle **text** as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See **assertHTMLEqual()** for more details.

SimpleTestCase.assertTemplateUsed(*response*, *template_name*, *msg_prefix=''*, *count=None*)[source]

Asserts that the template with the given name was used in rendering the response.

The name is a string such as **'admin/index.html'**.

The count argument is an integer indicating the number of times the template should be rendered. Default is **None**, meaning that the template should be rendered one or more times.

You can use this as a context manager, like this:

```
with self.assertTemplateUsed('index.html'):
    render_to_string('index.html')
with self.assertTemplateUsed(template_name='index.html'):
    render_to_string('index.html')
```

SimpleTestCase.assertTemplateNotUsed(*response*, *template_name*, *msg_prefix=''*)[source]

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

You can use this as a context manager in the same way as **assertTemplateUsed()**.

SimpleTestCase.assertRedirects(*response*, *expected_url*, *status_code=302*, *target_status_code=200*, *msg_prefix=''*, *fetch_redirect_response=True*)[source]

Asserts that the response returned a **status_code** redirect status, redirected to **expected_url** (including any **GET** data), and that the final page was received with **target_status_code**.

If your request used the **follow** argument, the **expected_url** and **target_status_code** will be the url and status code for the final point of the redirect chain.

If **fetch_redirect_response** is **False**, the final page won't be loaded. Since the test client can't fetch externals URLs, this is particularly useful if **expected_url** isn't part of your Django app.

Scheme is handled correctly when making comparisons between two URLs. If there isn't any scheme specified in the location where we are redirected to, the original request's scheme is used. If present, the scheme in **expected_url** is the one used to make the comparisons to.

Deprecated since version 1.9:
The **host** argument is deprecated, as redirections are no longer forced to be absolute URLs.

SimpleTestCase.assertHTMLEqual(*html1*, *html2*, *msg=None*)[source]

Asserts that the strings **html1** and **html2** are equal. The comparison is based on HTML semantics. The comparison takes following things into account:

- Whitespace before and after HTML tags is ignored.
- All types of whitespace are considered equivalent.
- All open tags are closed implicitly, e.g. when a surrounding tag is closed or the HTML document ends.

- Empty tags are equivalent to their self-closing version.
- The ordering of attributes of an HTML element is not significant.
- Attributes without an argument are equal to attributes that equal in name and value (see the examples).

The following examples are valid tests and don't raise any **`AssertionError`**:

```
self.assertHTMLEqual(
    '<p>Hello <b>world!</p>',
    '''<p>
        Hello   <b>world! <b/>
    </p>'''
)
self.assertHTMLEqual(
    '<input type="checkbox" checked="checked" id="id_accept_terms" />',
    '<input id="id_accept_terms" type="checkbox" checked>'
)
```

`html1` and **`html2`** must be valid HTML. An **`AssertionError`** will be raised if one of them cannot be parsed.

Output in case of error can be customized with the **`msg`** argument.

`SimpleTestCase.assertHTMLNotEqual`(*html1*, *html2*, *msg=None*)[source]

Asserts that the strings **`html1`** and **`html2`** are *not* equal. The comparison is based on HTML semantics. See **`assertHTMLEqual()`** for details.

`html1` and **`html2`** must be valid HTML. An **`AssertionError`** will be raised if one of them cannot be parsed.

Output in case of error can be customized with the **`msg`** argument.

`SimpleTestCase.assertXMLEqual`(*xml1*, *xml2*, *msg=None*)[source]

Asserts that the strings **`xml1`** and **`xml2`** are equal. The comparison is based on XML semantics. Similarly to **`assertHTMLEqual()`**, the comparison is made on parsed content, hence only semantic differences are considered, not syntax differences. When invalid XML is passed in any parameter, an **`AssertionError`** is always raised, even if both string are identical.

Output in case of error can be customized with the **`msg`** argument.

`SimpleTestCase.assertXMLNotEqual`(*xml1*, *xml2*, *msg=None*)[source]

Asserts that the strings **`xml1`** and **`xml2`** are *not* equal. The comparison is based on XML semantics. See **`assertXMLEqual()`** for details.

Output in case of error can be customized with the **`msg`** argument.

`SimpleTestCase.assertInHTML`(*needle*, *haystack*, *count=None*, *msg_prefix=''*)[source]

Asserts that the HTML fragment **`needle`** is contained in the **`haystack`** one.

If the **`count`** integer argument is specified, then additionally the number of **`needle`** occurrences will be strictly verified.

Whitespace in most cases is ignored, and attribute ordering is not significant. The passed-in arguments must be valid HTML.

`SimpleTestCase.assertJSONEqual`(*raw*, *expected_data*, *msg=None*)[source]

Asserts that the JSON fragments **`raw`** and **`expected_data`** are equal. Usual JSON non-significant whitespace rules apply as the heavyweight is delegated to the **`json`** library.

Output in case of error can be customized with the **`msg`** argument.

`SimpleTestCase.assertJSONNotEqual`(*raw*, *expected_data*, *msg=None*)[source]

Asserts that the JSON fragments **`raw`** and **`expected_data`** are *not* equal. See **`assertJSONEqual()`** for further details.

Output in case of error can be customized with the **`msg`** argument.

`TransactionTestCase.assertQuerysetEqual`(*qs*, *values*, *transform=repr*, *ordered=True*, *msg=None*)[source]

Asserts that a queryset **`qs`** returns a particular list of values **`values`**.

The comparison of the contents of **`qs`** and **`values`** is performed using the function **`transform`**; by default, this means that the **`repr()`** of each value is compared. Any other callable can be used if **`repr()`** doesn't provide a unique or helpful comparison.

By default, the comparison is also ordering dependent. If **`qs`** doesn't provide an implicit ordering, you can set the **`ordered`** parameter to **`False`**, which turns the comparison into a **`collections.Counter`** comparison. If the order is undefined (if the given **`qs`** isn't ordered and the comparison is against more than one ordered values), a **`ValueError`** is raised.

Output in case of error can be customized with the **msg** argument.

TransactionTestCase.assertNumQueries(*num*, *func*, **args*, ***kwargs*)**[source]**

Asserts that when **func** is called with ***args** and ****kwargs** that **num** database queries are executed.

If a **"using"** key is present in **kwargs** it is used as the database alias for which to check the number of queries. If you wish to call a function with a **using** parameter you can do it by wrapping the call with a **lambda** to add an extra parameter:

```
self.assertNumQueries(7, lambda: my_function(using=7))
```

You can also use this as a context manager:

```
with self.assertNumQueries(2):
    Person.objects.create(name="Aaron")
    Person.objects.create(name="Daniel")
```

Tagging tests

New in Django 1.10.

You can tag your tests so you can easily run a particular subset. For example, you might label fast or slow tests:

```
from django.test import tag

class SampleTestCase(TestCase):

    @tag('fast')
    def test_fast(self):
        ...

    @tag('slow')
    def test_slow(self):
        ...

    @tag('slow', 'core')
    def test_slow_but_core(self):
        ...
```

You can also tag a test case:

```
@tag('slow', 'core')
class SampleTestCase(TestCase):
    ...
```

Then you can choose which tests to run. For example, to run only fast tests:

```
$ ./manage.py test --tag=fast
```

Or to run fast tests and the core one (even though it's slow):

```
$ ./manage.py test --tag=fast --tag=core
```

You can also exclude tests by tag. To run core tests if they are not slow:

```
$ ./manage.py test --tag=core --exclude-tag=slow
```

`test --exclude-tag` has precedence over **`test --tag`**, so if a test has two tags and you select one of them and exclude the other, the test won't be run.

Email services

If any of your Django views send email using Django's email functionality, you probably don't want to send email each time you run a test using that view. For this reason, Django's test runner automatically redirects all Django-sent email to a dummy outbox. This lets you test every aspect of sending email – from the number of messages sent to the contents of each message – without actually sending the messages.

The test runner accomplishes this by transparently replacing the normal email backend with a testing backend. (Don't worry – this has no effect on any other email senders outside of Django, such as your machine's mail server, if you're running one.)

`django.core.mail.outbox`

During test running, each outgoing email is saved in **`django.core.mail.outbox`**. This is a simple list of all **`EmailMessage`** instances that have been sent. The **`outbox`** attribute is a special attribute that is created *only* when the **`locmem`** email backend is used. It doesn't normally exist as part of the **`django.core.mail`** module and you can't import it directly. The code below shows how to access this attribute correctly.

Here's an example test that examines **`django.core.mail.outbox`** for length and contents:

```
from django.core import mail
from django.test import TestCase

class EmailTest(TestCase):
    def test_send_email(self):
        # Send message.
        mail.send_mail(
            'Subject here', 'Here is the message.',
            'from@example.com', ['to@example.com'],
            fail_silently=False,
        )

        # Test that one message has been sent.
        self.assertEqual(len(mail.outbox), 1)

        # Verify that the subject of the first message is correct.
        self.assertEqual(mail.outbox[0].subject, 'Subject here')
```

As noted previously, the test outbox is emptied at the start of every test in a Django **`*TestCase`**. To empty the outbox manually, assign the empty list to **`mail.outbox`**:

```
from django.core import mail

# Empty the test outbox
mail.outbox = []
```

Management Commands

Management commands can be tested with the **`call_command()`** function. The output can be redirected into a **`StringIO`** instance:

```
from django.core.management import call_command
from django.test import TestCase
from django.utils.six import StringIO

class ClosepollTest(TestCase):
    def test_command_output(self):
        out = StringIO()
        call_command('closepoll', stdout=out)
        self.assertIn('Expected output', out.getvalue())
```

Skipping tests

The unittest library provides the **@skipIf** and **@skipUnless** decorators to allow you to skip tests if you know ahead of time that those tests are going to fail under certain conditions.

For example, if your test requires a particular optional library in order to succeed, you could decorate the test case with **@skipIf**. Then, the test runner will report that the test wasn't executed and why, instead of failing the test or omitting the test altogether.

To supplement these test skipping behaviors, Django provides two additional skip decorators. Instead of testing a generic boolean, these decorators check the capabilities of the database, and skip the test if the database doesn't support a specific named feature.

The decorators use a string identifier to describe database features. This string corresponds to attributes of the database connection features class. See **django.db.backends.BaseDatabaseFeatures** class for a full list of database features that can be used as a basis for skipping tests.

skipIfDBFeature(feature_name_strings*)[source]**

Skip the decorated test or **TestCase** if all of the named database features are supported.

For example, the following test will not be executed if the database supports transactions (e.g., it would *not* run under PostgreSQL, but it would under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipIfDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```

skipUnlessDBFeature(feature_name_strings*)[source]**

Skip the decorated test or **TestCase** if any of the named database features are *not* supported.

For example, the following test will only be executed if the database supports transactions (e.g., it would run under PostgreSQL, but *not* under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipUnlessDBFeature('supports_transactions')
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```

Learn More

About Django

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity statement

Get Involved

Join a Group

Contribute to Django

Submit a Bug

Report a Security Issue

Follow Us

GitHub

Twitter

News RSS

Django Users Mailing List

Language: **en**

Documentation version: **1.10**