Digital Career Institute

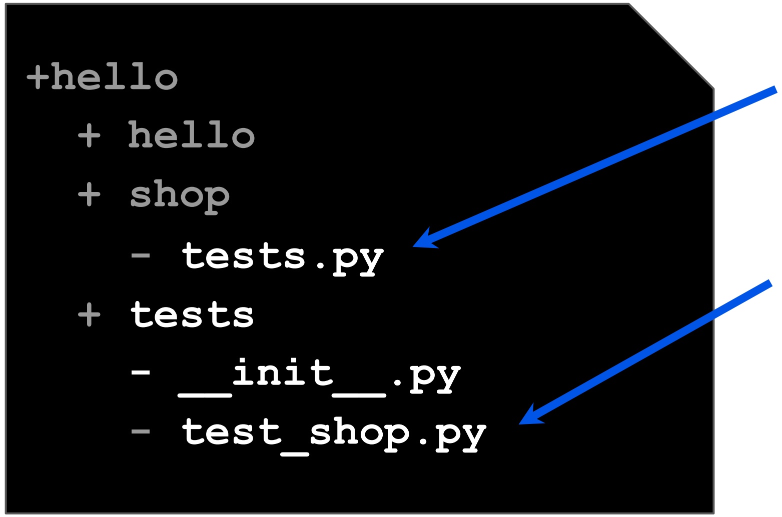
Python Course - Django Advanced Features

Digital Career Institute
DCI

Testing in Django

Digital Career Institute

Directory Tree



```
+hello
  + hello
  + shop
    - tests.py
  + tests
    - __init__.py
    - test_shop.py
```

Creating a Django app with `django-admin` (or `manage.py`) already creates a file named `tests.py` inside the app directory.

Another common approach is to create a tests folder on the root directory and add any amount of files in it.

Django will search any file in the directory tree whose name starts with `test` and will run any test defined in it.

Directory Tree

```
+hello
  + hello
  + shop
    - tests.py
  + tests
    - __init__.py
    - test_shop.py
```

Using one approach or the other is a personal choice.

If the Django app is designed as a Django package (installable on top of any Django project) the tests should be packed with the package.

If the Django app is just a feature in our site and is not meant to be installed on other Django projects, we may prefer to place all tests in one place.

Running Django Tests

```
(env) $ python manage.py test
System check identified no issues (0 silenced).

-------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Defining Unit Tests

Django uses the standard library module **unittest** that provides a class named **`TestCase`**.

shop/tests.py

```
from django.test import TestCase

class ShopTestCase(TestCase):
    def test_something(self):
        """Test a feature."""
        print("Testing something.")
    def test_something_else(self):
        """Test another feature."""
        print("Testing something else.")
```

Defining Unit Tests

```
(env) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Testing something
.Testing something else
.
-------------------------------------------------------
Ran 2 tests in 0.007s

OK
Destroying test database for alias 'default'...
```

Unit Tests Output

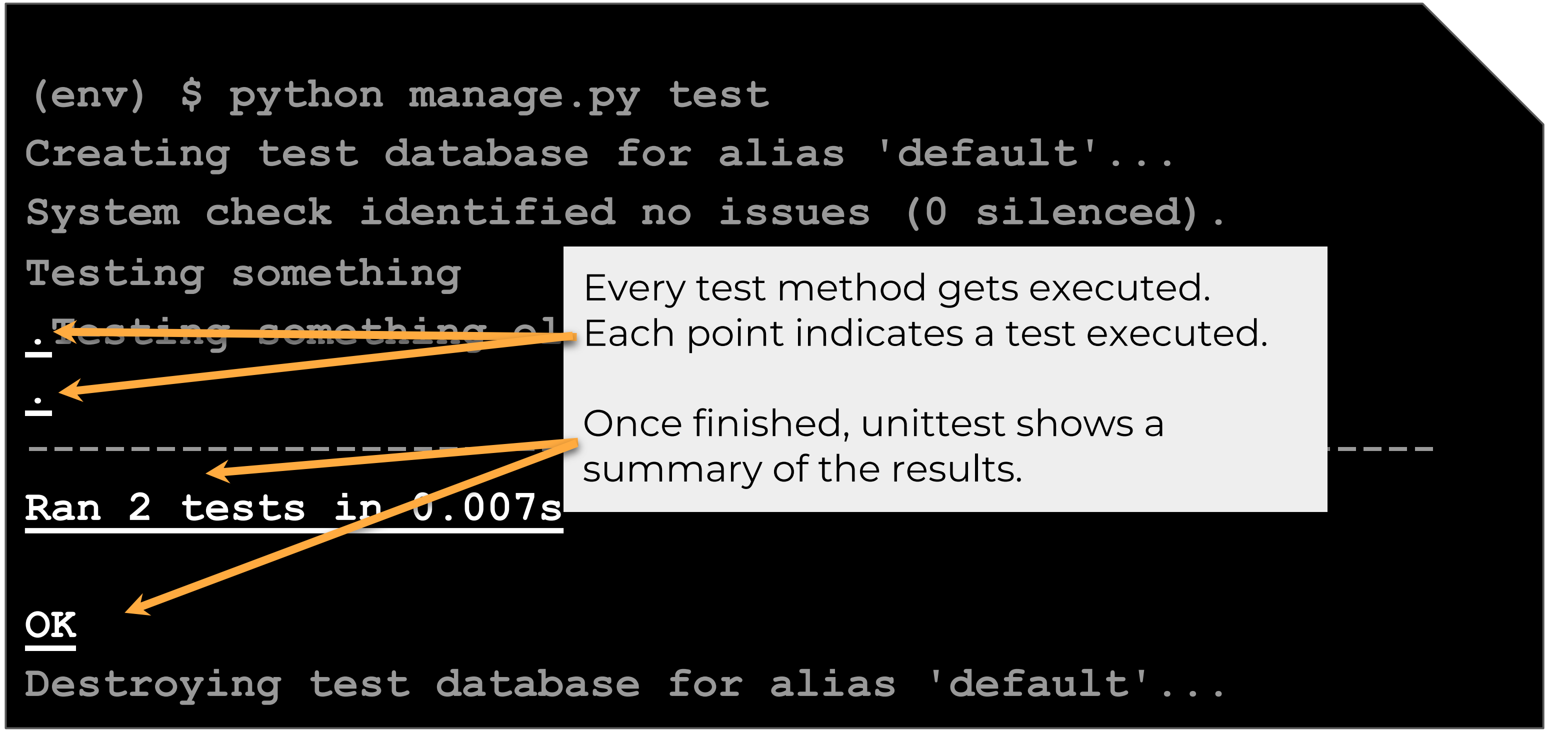
```
(env) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Testing something
.Testing something el
.
---------------------                              -----
Ran 2 tests in 0.007s

OK
Destroying test database for alias 'default'...
```

By default, Django creates a temporary database, so that tests who use it don’t impact the development database.

When tests are complete, it destroys the temporary database.

Unit Tests Output



```
(env) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Testing something
.Testing something el
.
----------------------                            -----
Ran 2 tests in 0.007s

OK
Destroying test database for alias 'default'...
```

Every test method gets executed.
Each point indicates a test executed.

Once finished, unittest shows a summary of the results.

Assertions

TEST	Test the contact form
ASSERTION 1	The subject field cannot be left empty.
ASSERTION 2	The section field cannot be left empty.
ASSERTION 3	A form with a section different than IT or Sales is invalid.

Assertions: assertEqual

shop/tests.py

```
from django.test import TestCase
from shop.app import ContactForm

class ShopTestCase(TestCase):
    def test_contact_form(self):
        """Test the contact form."""
        form = ContactForm()
        self.assertEqual(form.is_valid(), False)
```

The value of `form.is_valid()` must be equal to `False`.

Assertions: assertEqual

Digital Career Institute

shop/tests.py

```
from django.test import TestCase
from shop.app import ContactForm

class ShopTestCase(TestCase):
    def test_contact_form(self):
        """Test the contact form."""
        form = ContactForm()
        self.assertEqual(form.is_valid(), True)
```

```
Traceback (most recent call last):
  File ".../shop/tests.py", line 8, in test_contact_form
    self.assertEqual(form.is_valid(), True)
AssertionError: False != True
```

Assertions: assertTrue & assertFalse

shop/tests.py

```
from django.test import TestCase
from shop.app import ContactForm

class ShopTestCase(TestCase):
    def test_contact_form(self):
        """Test the contact form."""
        form = ContactForm()
        self.assertFalse(form.is_valid())
```

The value of `form.is_valid()` must be equal to `False`, so the shortcut `assertFalse` can also be used.

Assertions

As Django uses the module **unittest**, it has all the assertions provided by this module.

assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in

Choosing Tests to Run

```
(env) $ python manage.py test
(env) $ python manage.py test shop
(env) $ python manage.py test shop.tests
(env) $ python manage.py test shop.tests.ShopTestCase
(env) $ python manage.py test shop.tests.ShopTestCase.test_contact_form
(env) $ python manage.py test --pattern="test_shop*.py"
```

1. Run all tests.
2. Run all tests in the `shop` app.
3. Run all tests in the `tests` module of the `shop` app.
4. Run all tests in the `ShopTestCase` class of the `tests` module in the `shop` app.
5. Run the `test_contact_form` in `ShopTestCase` of the `tests` module in the `shop` app.
6. Run all tests in any Python file whose name starts with `test_shop`.

Choosing Tests to Run: Tags

shop/tests.py

```
from django.test import tag, TestCase
from shop.app import ContactForm

class ShopTestCase(TestCase):
    @tag("form", "contact")
    def test_contact_form(self):
        """Test the contact form."""
```

Multiple tags can be defined for each test.

```
(env) $ python manage.py test --tag=form
```

Run all tests tagged as `form` in any Python file anywhere on the directory tree.

Django Utility Tools: Client

shop/tests.py

```
from django.test import Client, TestCase

class ShopTestCase(TestCase):
    def test_login(self):
        """Test the login."""
        client = Client()
        response = client.get("/shop/")
        self.assertEqual(response.status_code, 302)
```

Client is an interface we can use to test client HTTP requests and responses.

Django Utility Tools: Client

shop/tests.py

```
from django.test import TestCase

class ShopTestCase(TestCase):
    def test_login(self):
        """Test the login."""
        response = self.client.get("/shop/")
        self.assertEqual(response.status_code, 302)
```

Django’s **`TestCase`** class already includes a **`Client`** instance in its **`client`** property.

Django Utility Tools: Client

shop/tests.py

```
from django.test import TestCase

class ShopTestCase(TestCase):
    def test_login(self):
        """Test the login."""
        ...
        response = self.client.get("/shop/", follow=True)
        print(response.redirect_chain)
        target = response.redirect_chain[0][0]
        self.assertEqual(target, "/shop/login/")
```

```
[('/shop/login/', 302)]
```

If the `follow` argument is set to `True`, it will execute the redirection and a property named `redirect_chain` will be available on the `response` object.

Django Utility Tools: Client

shop/tests.py

```
from django.test import TestCase

class ShopTestCase(TestCase):
    def test_login(self):
        """Test the login."""
        ...
        data = {"username": "a", "password": "b"}
        response = self.client.post("/shop/login/",
                                    data)
        ...
```

The **`Client`** instance can also be used to test HTTP POST requests.

Used with **`follow=True`**, will let us see exactly what happens in the **`redirect_chain`**.

Django Utility Tools: Client Response

shop/tests.py

```
from django.test import TestCase

class ShopTestCase(TestCase):
    def test_login(self):
        """Test the login."""
        ...
        response = self.client.get("/shop/")
        ...
```

The **`response`** object returned by the **`Client`**'s methods is not an **`HttpResponse`**.

Properties of **`response`**:

- client
  The Client instance.
- content
  The content returned.
- context
  The context used in the template rendering.
- json
  The JSON content as a dictionary.
- ...

Tapping the Test Control Flow

shop/tests.py

```
from django.test import TestCase

class ShopTestCase(TestCase):
    def setUp(self):
        """Run before each test."""
    def tearDown(self):
        """Run after each test."""
    @classmethod
    def setUpClass(cls):
        """Run before any test in this class."""
    @classmethod
    def tearDownClass(cls):
        """Run after all tests in this class."""
```

The unittest library provides methods that can be used to tap into the control flow.

We learned …

- That Django uses the Python module **unittest**.
- How to organize our test files in the directory tree.
- How to define and run tests using Django.
- How to run tests on forms and views.
- That we can use tags to choose what tests to run.
- How to execute instructions before and after the tests.

Digital Career Institute

Documentation

Documentation

Testing & Logging

- https://docs.djangoproject.com/en/4.2/topics/testing/
- https://docs.python.org/3/library/unittest.html

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute
DCI