HINTS

USANDO LOS MODELOS DE PRUEBA

Django utiliza el descubrimiento de pruebas integrado del módulo `unittest` (`built-in test discovery`), que descubrirá pruebas en el directorio de trabajo actual en cualquier archivo nombrado con el patrón `test*.py`. Siempre que se asigne un nombre a los archivos de forma adecuada, puede utilizar la estructura que desee. Recomendamos que cree un módulo para su código de prueba, y que éste tenga archivos separados para modelos, vistas, formularios, y cualquier otro tipo de código que necesite probar. Por ejemplo:

```
catalog/
  /tests/
    __init__.py
    test_models.py
    test_forms.py
    test_views.py
```

Cree una estructura de archivo, tal como se muestra arriba en su proyecto LocalLibrary. El `__init__.py` debe ser un archivo vacío (esto le dice a Python que el directorio es un paquete). Puede crear los tres archivos de prueba copiando y cambiando el nombre del archivo de prueba de esqueleto `/catalog/tests.py`.

Abre el archivo `/catalog/tests/test_models.py`. Éste debe importar `django.test.TestCase`, tal como se muestra:

```
from django.test import TestCase
# Create your tests here.
```

A menudo, agrega una clase de prueba para cada modelo / vista / formulario que desee probar, con métodos individuales para una funcionalidad específica. En otros casos, es posible que desee tener una clase separada para probar un caso de uso específico, con funciones individuales que prueben aspectos de ese caso de uso (por ejemplo: una clase para probar que un campo de modelo está validado correctamente, con funciones para probar cada uno de los posibles casos de falla).

Agregue la clase de prueba al final del archivo. Ésta demuestra cómo construir una clase de caso de prueba derivando de `TestCase`.

```python
class YourTestClass(TestCase):

    @classmethod
    def setUpTestData(cls):
        print("setUpTestData: Run once to set up non-modified data for all class methods.")
        pass

    def setUp(self):
        print("setUp: Run once for every test method to setup clean data.")
        pass

    def test_false_is_false(self):
        print("Method: test_false_is_false.")
        self.assertFalse(False)

    def test_false_is_true(self):
        print("Method: test_false_is_true.")
        self.assertTrue(False)

    def test_one_plus_one_equals_two(self):
        print("Method: test_one_plus_one_equals_two.")
        self.assertEqual(1 + 1, 2)
```

La nueva clase define dos métodos que puede utilizar para la configuración previa a la prueba (por ejemplo: para crear modelos, u otros objetos que necesitará).

`setUpTestData()` se llama una vez al comienzo de la ejecución de prueba para la configuración a nivel de clase. Usaría esto para crear objetos que no se modificarán ni cambiarán en ninguno de los métodos de prueba. `setUp()` se llama antes de cada función de prueba para configurar cualquier objeto que pueda ser modificado por la prueba (cada función de prueba obtendrá una versión "nueva" de estos objetos).

Las clases de prueba también tienen un método `tearDown()`, el cual no hemos utilizado. Éste no es particularmente útil para las pruebas de bases de datos, ya que en `TestCase` la clase base se encarga del desmontaje de la base de datos.

Debajo de ellos tenemos una serie de métodos de prueba, que utilizan funciones Assert to para probar si las condiciones son verdaderas, falsas, o iguales (`AssertTrue`, `AssertFalse`, `AssertEqual`). Si la condición no se evalúa como se esperaba, la prueba fallará y reportará el error a su consola.

Los `AssertTrue`, `AssertFalse`, `AssertEqual` son afirmaciones estándar proporcionadas por `unittest`. Hay otras aserciones estándar en el marco, y también unas específicas de Django para probar si una vista redirecciona (`assertRedirects`), si se ha utilizado una plantilla en particular (`assertTemplateUsed`), entre otros.

Generalmente, no debería incluir funciones `print()` en sus pruebas como se muestra arriba. Lo hacemos aquí solo para que pueda ver el orden en que se llaman las funciones de configuración en la consola.