Unit testing

Contents

Overview

Testing is a crucial part of software development. If we're going to develop Flask web applications, we need to know how to test them effectively.

This module discusses how to write unit tests for Flask applications that can be run by `pytest`.

TestBase Class

To test a Flask application, we have to create a `TestBase` class which allows us to define the conditions that the Flask application will be running under for each test.

`TestBase` inherits from the `TestCase` class provided by the `flask_testing` module, which can be installed with pip. We define the class like so:

```
from flask_testing import TestCase

class TestBase(TestCase):

    def create_app(self):
        # Defines the flask object's configuration for the unit tests

    def setUp(self):
        # Will be called before every test

    def tearDown(self):
        # Will be called after every test
```

There are three methods that we can define to configure the test conditions:

- `create_app()` - Defines the Flask configuration for the duration of the test.
- `setUp()` - Is run before each unit test. Often used to add dummy data to a database.
- `tearDown()` - Is run after each unit test. Often used to wipe the test database to prevent data from previous tests persisting, potentially causing false positives.

The unit tests themselves are written within classes that inherit from `TestBase`. For example:

```
class TestViews(TestBase):

    def test_home_get(self):
        response = self.client.get(url_for('home'))
        self.assertEqual(response.status_code, 200)

    def test_home_about(self):
        response = self.client.get(url_for('about'))
        self.assertEqual(response.status_code, 200)
```

The first unit test will send a `GET` request to the `home` route hosted the Flask application, then assert that the response should have a status code of `200` if the app is running correctly. It then does this for the `about` route.

We can define multiple classes that all inherit from `TestBase`. This is handy for organising our unit tests based on what they're testing, e.g. all methods testing Create functionality could be defined in a class named `TestCreate`, whereas methods testing Read functionality could be defined in a `TestRead` class.

All classes and their respective methods will be run when `pytest` is run.

`create_app()`

`create_app()` will return the Flask object that will be run for each unit test.

Typically we will import the `app` object from the Flask application like so:

```
from flask_testing import TestCase
from application import app

class TestBase(TestCase):

    def create_app(self):
        app.config.update(
            SQLALCHEMY_DATABASE_URI='sqlite:///',
            DEBUG=True,
            WTF_CSRF_ENABLED=False
        )
        return app
```

Using the method `app.config.update()` we can update the app's configuration for the tests.

For example, if we're testing CRUD functionality with a database, we should configure the application to interact with a *test database* rather than our production database, otherwise our test information may be visible to users using the application.

> Note: Testing applications that use WTForms can cause issues with CSRF form validation, causing the `validate_on_submit()` method to always return `False`.
>
> To circumvent this issue, you can disable CSRF protection for the tests by setting `WTF_CSRF_ENABLED=False`.

Here are some recommended configurations for your tests:

Configuration	Recommended Value	Effect
SQLALCHEMY_DATABASE_URI	sqlite:///	Tests use an in-memory database
WTF_CSRF_ENABLED	False	Disables CSRF protection

Configuration	Recommended Value	Effect
DEBUG	True	Ensures the application provides debugging information when errors occur

`setUp()`

The `setUp()` method will be run before each test. This allows us to define some 'environmental' conditions for each test.

Commonly, we use `setUp()` to configure our database and add test information:

```
def setUp(self):
    # Create table schema
    db.create_all()

    # Create test dog
    test_dog = Dog(name="Chewbarka")

    # save sample data to database
    db.session.add(test_dog)
    db.session.commit()
```

This makes sure that the database's schema has been created and some dummy information has been inserted into a table (in this case, a `Dog` table).

Now, if we want to test Read functionality of our application, we can do so by testing the relevant function to check if the dog "Chewbarka" is returned.

If we didn't use `setUp()`, we would first have to get the app to Create some data, which would depend the Create functionality to be working correctly. This goes against the design philosophy of unit tests, which should be written to test one 'unit' of functionality at a time.

`tearDown()`

The `tearDown()` method is run after each test and is used to reset the testing environment before the next unit test is started.

Typically we would use this to delete the contents of the test database, such as with the following code:

```
def tearDown(self):
    db.session.remove()
    db.drop_all()
```

This prevents data created in a previous unit test from being persisted to a proceeding unit test, creating a blank slate for each test.

Doing so prevents our tests from returning false postive or negative results. To illustrate this, consider the following test script:

```
class TestBase(TestCase):
    def create_app(self):
        app.config.update(SQLALCHEMY_DATABASE_URI='sqlite:///')
        return app

    def setUp(self):
        db.create_all()
        test_dog = Dog(name="Chewbarka")
        db.session.add(test_dog)
        db.session.commit()

class TestAdd(TestBase):
    def test_add_dog(self):
        # add a new dog to the database
        response = self.client.post(
            url_for('add_dog'),
            data = dict(name="Barkus Aurelius")
        )
        # test that the newly-added dog has an id value of 2,
        # as it's the second dog that's been added to the database
        # after Chewbarka in the setUp() method
        assert Dog.query.filter_by(name="Barkus Aurelius").id == 2

class TestDelete(TestBase):
    def test_delete_dog(self):
        # delete Chewbarka from the database
        response = self.client.delete(
            url_for('delete_dog'),
            data = dict(name="Chewbarka")
        )
        # query the dog table - if Chewbarka has been deleted,
        # the query should return an empty list
        assert len(Dog.query.all()) == 0
```

Currently, the `test_delete_dog()` test will fail, because when the test queries the `Dog` table it will retrieve a dog named "Barkus Aurelius" who had been created in the `test_add_dog()` test. This results in a **false negative** result, where the function being test may work but will always fail the test.

By adding the `tearDown()` method to the `TestBase` class as below, it will ensure that "Barkus Aurelius" has been wiped from the database.

```
def tearDown(self):
    db.session.remove()
    db.drop_all()
```

Making HTTP Requests

We can use the `client` subclass of TestCase to execute requests in tests.

You can make a `GET` request with `client.get(<URL>)`:

```
response = self.client.get(url_for('home'))
```

You can make a `POST` request with `client.post(<URL>)`:

```
response = self.client.post(
    url_for('add_dog'),
    data = dict(name="Barkus Aurelius")
)
```

We can provide information that needs to be sent with a `POST` request using the `data` parameter. This is useful for testing forms.

If you are expecting the page to redirect to another location after completing the request, you need to provide an extra argument with `follow_redirects=True`:

```
response = self.client.post(
    url_for('add_dog'),
    data = dict(name="Barkus Aurelius"),
    follow_redirects=True
)
```

Each of these methods will return the HTTP response as an object, allowing us to inspect its contents.

Assertions

`TestCase` provides a variety of assert methods (with their opposite counterparts) for different logic checks. For example:

```
class TestViews(TestBase):
    def test_home_get(self):
        response = self.client.get(url_for('home')) # send a GET request
        self.assertEqual(response.status_code, 200) # assert that the response
code is 200
        self.assertIn(b'Welcome to my website', response.data) # assert that the
website's title is present in the HTTP response's data
```

Here are a few assertions provided by `TestCase`:

Method	Checks	Opposite
`assertEqual(a,b)`	a == b	`assertNotEqual(a,b)`
`assertTrue(x)`	x is True	`assertFalse(X)`
`assertIs(a,b)`	a is b	`assertIsNot(a,b)`
`assertIsNone(x)`	x is None	`assertIsNotNone(x)`
`assertIn(a,b)`	a in b	`assertNotIn(a,b)`
`assertIsInstance(a,b)`	a is an instance of b	`assertNotIsInstance(a,b)`
`assert<status_code>`	Checks the HTTP response code where `<status_code>` should be replaced with a status code number, for example `assert200` or `assert502`	N/A

Tutorial

In this tutorial we will conduct tests on a very simple Flask web application.

Setup

This tutorial assumes you are working on an Ubuntu VM, at least version `18.04 LTS`.

First, install apt dependencies:

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
```

Create a directory named `flask-unit-testing` and make it your current working directory:

```
mkdir flask-unit-testing && cd $_
```

We now need to create a Python virtual environment to install our pip requirements in. Create a new virtual environment named `venv` and activate it:

```
python3 -m venv venv
source venv/bin/activate
```

Creating the App

We are going to create the following file structure:

```
├──templates
|  └──home.html
├──tests
|  ├──__init__.py
|  └──test_app.py
├──app.py
└──requirements.txt
```

Run these commands to create it:

```
mkdir templates tests
touch app.py requirements.txt \
    templates/home.html \
    tests/{__init__,test_app}.py
```

Insert the following into `requirements.txt`:

```
flask
flask_sqlalchemy
flask_wtf
flask_testing
pytest
pytest-cov
```

Install the pip requirements by entering:

```
pip3 install -r requirements.txt
```

Insert the following into `app.py`:

```
from flask import Flask, redirect, render_template, request
from flask_sqlalchemy import SQLAlchemy
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///data.db"
app.config['SECRET_KEY'] = "shhh it's a secret"

db = SQLAlchemy(app)

class Register(db.Model):
    name = db.Column(db.String(30), nullable=False, primary_key=True)

class RegisterForm(FlaskForm):
    name = StringField('Name')
    submit = SubmitField('Submit')

db.create_all()

@app.route('/', methods=["GET","POST"])
def home():
    form = RegisterForm()
    if form.validate_on_submit():
        person = Register(name=form.name.data)
        db.session.add(person)
        db.session.commit()
    registrees = Register.query.all()
    return render_template("home.html", registrees=registrees, form=form)

if __name__ == '__main__':
    app.run(port=5000, debug=True, host='0.0.0.0')
```

This is a simple application that prompts the user to enter their name, then adds them to a database.

Next, insert the following code into `templates/home.html`:

```
<html>
  <head>
    <title>Add person</title>
  </head>
  <body>
    <h1>Add Person</h1>
    <form method="POST" action="/">
        {{ form.hidden_tag() }}
        {{ form.name.label }} {{ form.name }}
        {{ form.submit }}
    </form>
    <h2>Register</h2>
    <table>
      {% for person in registrees %}
        <tr>
          <td>
            {{ person.name }}
          </td>
        </tr>
        {% endfor %}
    </table>
  </body>
</html>
```

This HTML template will display the form and list any registrees found in the database underneath.

Create the Unit Tests

Now we are going to create our unit test script.

Insert the following into `test_app.py`:

```
# Import the necessary modules
from flask import url_for
from flask_testing import TestCase

# import the app's classes and objects
from app import app, db, Register

# Create the base class
class TestBase(TestCase):
    def create_app(self):

        # Pass in testing configurations for the app.
        # Here we use sqlite without a persistent database for our tests.
        app.config.update(SQLALCHEMY_DATABASE_URI="sqlite:///",
                SECRET_KEY='TEST_SECRET_KEY',
                DEBUG=True,
                WTF_CSRF_ENABLED=False
                )
        return app

    # Will be called before every test
    def setUp(self):
        # Create table
        db.create_all()
        # Create test registree
        sample1 = Register(name="MsWoman")
        # save users to database
        db.session.add(sample1)
        db.session.commit()

    # Will be called after every test
    def tearDown(self):
        # Close the database session and remove all contents of the database
        db.session.remove()
        db.drop_all()

# Write a test class to test Read functionality
class TestViews(TestBase):
    def test_home_get(self):
        response = self.client.get(url_for('home'))
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'MsWoman', response.data)
```

Running the Unit Tests

Now we are ready to run the tests.

Execute the test and include a coverage report by running:

```
python3 -m pytest --cov
```

This will gather coverage of our app, our test code and any code in the `venv`, which will skew our coverage result.

We can test the coverage of just the `app.py` file by running:

```
python3 -m pytest --cov=app
```

You should see that the tests currently only achieve `87%` coverage.

We can see which lines of code we haven't been able to test by running:

```
python3 -m pytest --cov=app --cov-report term-missing
```

Can you figure out why those lines aren't being covered by the test?

We can also save the coverage report to an HTML file by running the following

```
python3 -m pytest --cov=app --cov-report html
```

This will create a directory called `htmlcov`. There are many files in there but the one we are interested in is `index.html`. Open that in a web browser to access the coverage report.

Clean Up

To stop your Flask application running, navigate back to your terminal and press `Ctrl+C`. You should now have control over your terminal again.

To deactivate the virtual environment, run:

```
deactivate
```

If you wish to delete the virtual environment, run:

```
rm -rf venv
```

Exercises

The Tutorial's unit test script currently doesn't Create functionality for the application. Write a unit test that tests this functionality by adding a new registree to the database. Doing so should provide you with 100% coverage.

▶ Hint