Testing

Download Demo Code <../flask-testing-demo.zip>

Goals

- Discuss the benefits of writing tests
- Compare unit, integration, and end-to-end tests
- Compare different ways to write tests in Python:
  - ***assert*** statements
  - doctests
  - ***unittest*** module
- Write integration tests for our Flask apps
- Use tests to inform how we write application code

Whys and Wherefores

Can't I Just Test Code Myself?

Yes. You probably do so now.

Testing is

- #1 thing employers ask us about
- Something ALL engineers do
- Automating the boring stuff
- Fascinating and highly skilled art
- Peace of mind: develop with confidence!

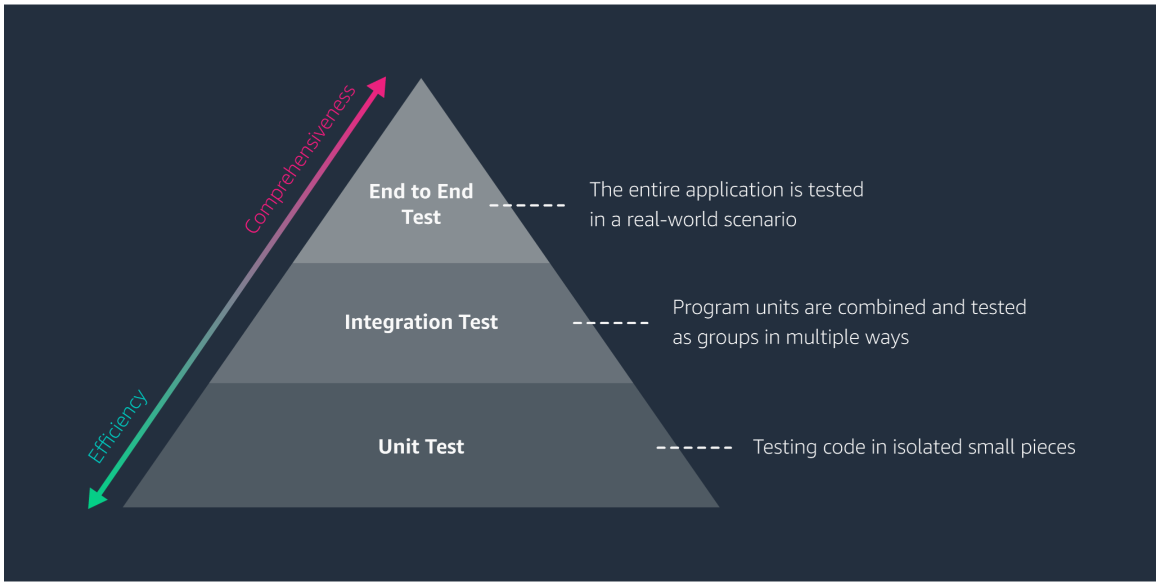
Automated Tests Are Particularly Good For

- Testing things that "should work"
- Testing the edge cases (anticipate the unexpected)
- New things don't break things that were working ("regression")

Kinds of Tests

Testing a Dryer

- **Unit Test**: does this individual component work?
- **Integration Tests**: do the parts work together?
- **End-to-end Test**: wet clothes → dry clothes?


Comprehensiveness
Efficiency
End to End Test
The entire application is tested in a real-world scenario
Integration Test
Program units are combined and tested as groups in multiple ways
Unit Test
Testing code in isolated small pieces
<_images/testing-pyramid.png>

Some people call and include other notions of testing levels, like “acceptance tests”, “system tests”, and others.

Unit Tests

- Test one “unit” of functionality
  - Typically, one function or method
- Don’t test integration of components
  - Don’t test framework itself *(eg, Flask)*
- Promote modular code
  - Write code with testing in mind

You Can Do This By Hand

```
def adder(x, y):
    """Add two numbers together."""
    print("INSIDE ADDER!")
    return x + y
```

```
assert adder(2, 5) == 7
assert adder(2, 7) == 10, "expected 2+7 to be 10"
assert adder(2, 3) == 5

print("HELLO WORLD!")
```

- ***assert*** raises ***AssertionError*** if expression is False
- Can provide optional exception message
- Code exits as soon as an exception is raised

DocTests

DocTests are awesome!

"Testable documentation"

"Documented testing"

doctest module in Python standard library

Our Adder

```
def adder(x, y):
    """Add two numbers together."""
    print("INSIDE ADDER!")
    return x + y
```

Let's try it out!

```
$ python
>>> from arithmetic import adder
>>> adder(1, 1)
2
>>> adder(-1, 1)
0
```

```
def adder(x, y):
    """Adds two numbers together.

        >>> adder(1, 1)
        2
        >>> adder(-1, 1)
        0
    """

    return x + y
```

Running DocTests

```
$ python -m doctest arithmetic.py
$ (nothing output for success)
```

Everything worked!

Running Verbosely

```
$ python -m doctest -v arithmetic.py
Trying:
    adder(1, 1)
Expecting:
    2
ok
Trying:
    adder(-1, 1)
Expecting:
    0
ok
1 items had no tests:
    arithmetic
1 items passed all tests:
2 tests in arithmetic.adder
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Let's Make it Fail

```
def adder(x, y):
    """Adds two numbers together.

        >>> adder(1, 1)
        2

        >>> adder(-1, 1)
        0
    """
    return x + y + 1  # this is wrong
```

Running DocTests

```
$ python -m doctest arithmetic.py
**********************************************************************
File "arithmetic.py", line 10, in arithmetic.adder
Failed example:
    adder(1, 1)
Expected:
    2
Got:
    3
**********************************************************************
File "arithmetic.py", line 15, in arithmetic.adder
Failed example:
    adder(-1, 1)
Expected:
    0
Got:
    1
**********************************************************************
1 items had failures:
   2 of   2 in arithmetic.adder
*Test Failed* 2 failures.
```

Note: DocTest options

You can also add special comments in your doctests to say "be a little less strict about how the output matches".

For example, sometimes you might have so much output it would be overwhelming to show it all:

```
>>> range(16)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

By using the `ELLIPSIS` option, you can elide part of that in your test, like this:

```
>>> print range(16)
[0, 1, ..., 14, 15]
```

Or, if your output may have awkward linebreaks and whitespace that might make it hard to use in a test, you can use `NORMALIZE_WHITESPACE` to ignore all whitespace differences between the expected output and the real output:

```
>>> poem_line
```

```
'My father moved through
dooms of love'
```

Python unittest module

unittest module

Unit testing via classes! In the Python standard library.

demo/test_arithmetic.py

```
import arithmetic
from unittest import TestCase


class AdditionTestCase(TestCase):
    """Examples of unit tests."""

    def test_adder(self):
        assert arithmetic.adder(2, 3) == 5
```

- Test cases are a bundle of tests
  - In a class that subclasses `TestCase`
  - Test methods **must** start with `test_`
- `python -m unittest NAME_OF_FILE` runs all cases

TestCase assertions

demo/test_arithmetic.py

```
class AdditionTestCase(TestCase):
    """Examples of unit tests."""

    def test_adder(self):
        assert arithmetic.adder(2, 3) == 5

    def test_adder_2(self):
        # instead of assert arithmetic.adder(2, 2) == 4
        self.assertEqual(arithmetic.adder(2, 2), 4)
```

- Provides better output, including expected value

Method	Checks that
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

Method	Checks that
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

DocTest or unittest Class?

- DocTests keep tests close to code
  - Too many tests can drown out code
- ***unittest*** classes good for when you have lots of tests
  - Or interesting hierarchies of tests

Integration Tests

Test that components work together

Integration Testing Flask App

What kinds of things do we want to test in our Flask applications?

- "Does this URL path map to a route function?"
- "Does this route return the right HTML?"
- "Does this route return the correct status code?"
- "After a POST to this route, are we redirected?"
- "After this route, does the session contain expected info?"

Writing Integration Tests

You write them with ***unittest*** framework.

Yeah, I know. Weird.

test_client

demo/test_app.py

```
from app import app
```

demo/test_app.py

```
class ColorViewsTestCase(TestCase):
    """Examples of integration tests: testing Flask app."""

    def test_color_form(self):
        with app.test_client() as client:
            # can now make requests to flask via `client`
```

Technically, this comes from “Werkzeug”, a library that Flask uses.

This doesn’t start a real web server — but we can make requests to Flask via ***client***.

GET Request

demo/test_app.py

```
def test_color_form(self):
    with app.test_client() as client:
        # can now make requests to flask via `client`
        resp = client.get('/')
        html = resp.get_data(as_text=True)

        self.assertEqual(resp.status_code, 200)
        self.assertIn('<h1>Color Form</h1>', html)
```

POST and Form Data

demo/test_app.py

```
def test_color_submit(self):
    with app.test_client() as client:
        resp = client.post('/fav-color',
                           data={'color': 'blue'})
        html = resp.get_data(as_text=True)

        self.assertEqual(resp.status_code, 200)
        self.assertIn('Woah! I like blue, too', html)
```

Testing Redirects

demo/test_app.py

```
def test_redirection(self):
    with app.test_client() as client:
        resp = client.get("/redirect-me")
        self.assertEqual(resp.status_code, 302)
        self.assertEqual(resp.location, "http://localhost/")
```

`follow_redirects=True` makes new request when response redirects:

demo/test_app.py

```
def test_redirection_followed(self):
    with app.test_client() as client:
        resp = client.get("/redirect-me", follow_redirects=True)
        html = resp.get_data(as_text=True)

        self.assertEqual(resp.status_code, 200)
        self.assertIn('<h1>Color Form</h1>', html)
```

Testing the Session

To test value of session:

demo/test_app.py

```
from flask import session
```

demo/test_app.py

```
def test_session_info(self):
    with app.test_client() as client:
        resp = client.get("/")

        self.assertEqual(resp.status_code, 200)
        self.assertEqual(session['count'], 1)
```

To set the session before the request, add block like this:

demo/test_app.py

```
    def test_session_info_set(self):
        with app.test_client() as client:
            # Any changes to session should go in here:
            with client.session_transaction() as change_session:
                change_session['count'] = 999

            # Now those changes will be in Flask's `session`
            resp = client.get("/")

            self.assertEqual(resp.status_code, 200)
            self.assertEqual(session['count'], 1000)
```

setUp and *tearDown*

setUp and ***tearDown*** methods are called before/after *each test*.

```
class FlaskTests(TestCase):

  def setUp(self):
      """Stuff to do before every test."""

  def tearDown(self):
      """Stuff to do after each test."""

  def test_1(self):
      ...

  def test_2(self):
      ...
```

Runs, in order: ***setUp***, ***test_1***, ***tearDown setUp***, ***test_2***, ***tearDown***

Often useful to add/remove data in test database before/after each test

Making Testing Easier

Add these before test case classes:

demo/test_app.py

```
# Make Flask errors be real errors, not HTML pages with error info
app.config['TESTING'] = True

# This is a bit of hack, but don't use Flask DebugToolbar
app.config['DEBUG_TB_HOSTS'] = ['dont-show-debug-toolbar']
```

Note: Seeing Errors In Tests

If a route raises an error, it can be hard to debug this in a test.

For example, in your ***server.py***:

```
@app.route('/')
def homepage():
    raise KeyError("Foo")
```

In your ***test_app.py***:

```
class MyTest(unittest.TestCase):
    def test_home(self):
        client = app.test_client()

        result = client.get('/')
        self.assertEqual(result.status_code, 200)
```

When you run your tests, it will fail, as that route returns a 500 (Internal Server Error), not a 200

(Ok). However, you won't see the error message of the server.

To fix this, you can set the Flask app's configuration to be a in ***TESTING*** mode, and it will print all Flask errors to the console:

This what `app.config['TESTING'] = True` does.

Breaking Down Code

Intermixed Concerns

How do we test this?

```
@app.route('/taxes', methods=['POST'])
def taxes():
    """Calculate taxes from web form."""

    income = request.form.get('income')

    # Calculate the taxes owed
    owed = income / 45.3 * random.randint(100) / other_stuff

    return render_template("taxes.html", owed=owed)
```

Breaking Down Code

Very often, you'll want to separate web interface from logic

```
def calculate_taxes(income):
    """Calculate taxes owed for this income."""

    ...

@app.route('/taxes', methods=['POST'])
def taxes():
    """Calculate taxes from web form."""

    income = request.form.get('income')
    owed = calculate_taxes(income)

    return render_template("taxes.html", owed=owed)
```

How Many Tests??

- Ask yourself: is there too much logic in your view function?

- When you test, you don't need one assertion per test function
- Remember to test failing things, like forms that don't validate

Organizing / Running Tests

Small Projects

For small projects, keep tests in one file, ***tests.py***:

```
├── app.py
├── requirements.txt
└── tests.py
```

Run them like this:

```
(venv) $ python -m unittest
```

Larger Projects

For more complex projects, organize in files named ***test_something.py***:

```
├── app.py
├── requirements.txt
├── test_cats.py
└── test_dogs.py
```

Run all of them like this:

```
(venv) $ python -m unittest
```

Can also run individual files / cases / test methods:

```
(venv) $ python -m unittest test_cats

(venv) $ python -m unittest test_cats.CatViewTestCase

(venv) $ python -m unittest test_cats.CatViewTestCase.test_meow
```

Looking Ahead

Resources

- Doctests: https://docs.python.org/2/library/doctest.html <https://docs.python.org/2/library/doctest.html>
- Unittest: https://docs.python.org/2/library/unittest.html <https://docs.python.org/2/library/unittest.html>
- Flask Testing http://flask.pocoo.org/docs/1.0/testing/ <http://flask.pocoo.org/docs/1.0/testing/>
- Test Client http://werkzeug.pocoo.org/docs/0.14/test/ <http://werkzeug.pocoo.org/docs/0.14/test/>

Future Topics

- Unit & Integration Testing for JS
- End-to-end Tests
  - Does it work? In a real browser? For real?
- “Test-driven development”