# Intermediate Testing

# Fixtures

As you write more tests, you'll create test case classes whose methods start or end with the same lines of code.

Consolidate these in `setUp()` and `tearDown()`.

```
class TestSample(unittest.TestCase):
    def setUp(self):
        "Run before every test methods starts."
        self.conn = connect_to_test_database()
    def tearDown(self):
        "Run after every test method ends."
        self.conn.close()
    def test_create_tables(self):
        from mylib import create_tables
        create_tables(self.conn)
```

**Watch Out**: It's "setUp", not "setup". "tearDown", not "teardown".

# Example

Imagine writing a program that saves it state between runs. It saves it to a special file, called the "state file".

```
# statefile.py
class State:
    def __init__(self, state_file_path):
        # Load the stored state data, and save
        # it in self.data.
        self.data = { }
    def close(self):
        # Handle any changes on application exit.
```

# Planning The Test

Tests on the `State` class should verify:

- If you add a new key-value pair to the state, it is recorded correctly in the state file.
- If you alter the value of an existing key, that updated value is written to the state file.
- If the state is not changed, the state file's content stays the same.

# test_statefile: initial code

```
# test_statefile.py
import os
import unittest
import shutil
import tempfile
from statefile import State

INITIAL_STATE = '{"foo": 42, "bar": 17}'
```

# test_statefile: setUp and tearDown

```
class TestState(unittest.TestCase):
    def setUp(self):
        self.testdir = tempfile.mkdtemp()
        self.state_file_path = os.path.join(
            self.testdir, 'statefile.json')
        with open(self.state_file_path, 'w') as outfile:
            outfile.write(INITIAL_STATE)
        self.state = State(self.state_file_path)

    def tearDown(self):
        shutil.rmtree(self.testdir)
```

# test_statefile: the tests

```
    def test_change_value(self):
        self.state.data["foo"] = 21
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertEqual(21,
            reloaded_statefile.data["foo"])

    def test_remove_value(self):
        del self.state.data["bar"]
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertNotIn("bar", reloaded_statefile.data)

    def test_no_change(self):
        self.state.close()
        with open(self.state_file_path) as handle:
            checked_content = handle.read()
        self.assertEqual(checked_content, INITIAL_STATE)
```

# Expecting Exceptions

Sometimes your code is *supposed* to raise an exception. And it's an error if, in that situation, it does not.

Use `TestCase.assertRaises()` to verify.

Imagine a `roman2int()` function:

```
>>> roman2int("XVI")
16
>>> roman2int("II")
2
>>> roman2int("a thousand")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in roman2int
ValueError: Not a roman numeral: a thousand
```

# Asserting Exceptions

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("bad value")
```

# Catching The Error

## If `roman2int()` does NOT raise `ValueError`:

```
$ python3 -m unittest test_roman2int.py
F
======================================================
FAIL: test_roman2int_error (test_roman2int.TestRoman)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_roman2int.py", line 7, in test_roman2int_error
    roman2int("bad value")
AssertionError: ValueError not raised

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

# Inspecting Exceptions

You can also make assertions on the exception object itself. To do this, capture its *context* with an "as" clause:

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError) as context:
            roman2int("bad value")
        exception = context.exception
        expected_message = "Not a roman numeral: bad value"
        actual_message = exception.args[0]
        self.assertEqual(expected_message, actual_message)
```

# Homework: Intermediate Unit Tests

## Instructions: `lab-intermediate.txt`

- In labs/py3 for 3.x; labs/py2 for 2.7
- First follow the instructions to write `textlib.py` and `test_textlib.py`.
- Optional extra credit instructions in `lab-intermediate.txt`.