Test-Driven Development

Automated tests

An *automated test* is a program that tests another program.

Some of you have some experience with this already.

There are different flavors: unit tests, integration tests, etc.

We'll focus on the common basis, looking mainly at the simpler unit tests.

Why Write Tests?

It's basically a superpower.

Writing automated tests is one of the keys that separate average developers from world-class software engineers.

The ceiling of software complexity you can gracefully handle is *several quantum leaps higher* once you master unit tests.

This is well worth your while.

A Simple Test

Let’s write an automated test for this function:

```
# Split a number into portions, as evenly as possible. (But it has a bug.)
def split_amount(amount, n):
    portion, remain = amount // n, amount % n
    portions = []
    for i in range(n):
        portions.append(portion)
        if remain > 1:
            portions[-1] += 1
            remain -= 1
    return portions
```

How it ought to work:

```
>>> split_amount(4, 2)
[2, 2]
>>> split_amount(5, 3)
[2, 2, 1]
```

The Test Function

Here's a function that will test it:

```
def test_split_amount():
    assert [1] == split_amount(1, 1)
    assert [2, 2] == split_amount(4, 2)
    assert [2, 2, 1] == split_amount(5, 3)
    assert [3, 3, 2, 2, 2] == split_amount(12, 5)
    print("All tests pass!")
# And of course, invoke it.
test_split_amount()
```

If any assertions fail, you'll see a stack trace:

```
Traceback (most recent call last):
  File "demo1.py", line 22, in <module>
    test_split_amount()
  File "demo1.py", line 18, in test_split_amount
    assert [2, 2, 1] == split_amount(5, 3)
AssertionError
```

Detecting the Error

The assertion that failed is:

```
assert [2, 2, 1] == split_amount(5, 3)
```

The good: Tells you an input that breaks the function.

The bad: Doesn’t tell you anything else.

- What was the incorrect output?
- What other tests fail? The testing stops immediately, even if there are other assertions.
- Your large applications will have MANY tests. How do you reliably make sure you’re running them all?
- Can we improve on the *reporting* of the test results?
- What about different assertion types?

Python’s `unittest` module solves all these problems.

import unittest

Here’s a basic unit test.

```
# test_splitting.py

from unittest import TestCase
from splitting import split_amount

class TestSplitting(TestCase):
    def test_split_amount(self):
        self.assertEqual([1], split_amount(1, 1))
        self.assertEqual([2, 2], split_amount(4, 2))
        self.assertEqual([2, 2, 1], split_amount(5, 3))
        self.assertEqual([3, 3, 2, 2, 2], split_amount(12, 5))
```

Running The Test

```
$ python3 -m unittest test_splitting.py
F
======================================================================
FAIL: test_split_amount (test_splitting.TestSplitting)
---------------------------------------
Traceback (most recent call last):
  File "test_splitting.py", line 8, in test_split_amount
    self.assertEqual([2, 2, 1], split_amount(5, 3))
AssertionError: Lists differ: [2, 2, 1] != [2, 1, 1]
First differing element 1:
2
1

- [2, 2, 1]
?     ^
+ [2, 1, 1]
?     ^
---------------------------------------
Ran 1 test in 0.001s
FAILED (failures=1)
```

Corrected Function

```python
def split_amount(amount, n):
    'Split an integer amount into portions, as even as possible.'
    portion, remain = amount // n, amount % n
    portions = []
    for i in range(n):
        portions.append(portion)
        if remain > 0: # Was "remain > 1"
            portions[-1] += 1
            remain -= 1
    return portions
```

```
$ python3 -m unittest test_splitting.py
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

What’s happening?

```
python3 -m unittest test_splitting.py
```

`unittest` is a standard library module. `test_splitting.py` is the file containing tests.

Inside is a class called `TestSplitting`. It subclasses `TestCase`.

(The name doesn’t have to start with "Test", but often will.)

It has a method named `test_split_amount()`. That *test method* contains assertions.

Test methods **must** start with the string "test", or they won’t get run.

Test Modules

To run code in a specific file:

```
python3 -m unittest test_splitting.py
```

OR a module name:

```
python3 -m unittest test_splitting
```

`test_splitting` is a **module**. It can be implemented as one or many files, just like any module.

In Python 2, you **must** pass the module argument, NOT the filename.

Test Discovery

You can also just run:

```
python3 -m unittest
```

This will locate all test code under the current directory.

This is called **test discovery**.

Restriction: the module/filename **must** start with "test" to be discovered.

To see options, run with `-h`:

```
python3 -m unittest -h
```

Assertions

`TestSplitting` uses the `assertEqual` method.

```python
class TestSplitting(TestCase):
    def test_split_amount(self):
        self.assertEqual([1], split_amount(1, 1))
        self.assertEqual([2, 2], split_amount(4, 2))
        self.assertEqual([2, 2, 1], split_amount(5, 3))
        self.assertEqual([3, 3, 2, 2, 2], split_amount(12, 5))
```

Notice the expected value is always first. Consistency.

You can also make it always second. Just don't alternate in the same codebase.

Other Assertions

There are many different assertion methods. You'll most often use `assertEqual`, `assertNotEqual`, `assertTrue`, and `assertFalse`.

```
class TestDemo(TestCase):
    def test_assertion_types(self):
        self.assertEqual(2, 1 + 1)
        self.assertNotEqual(5, 1 + 1)
        self.assertTrue(10 > 1)
        self.assertFalse(10 < 1)
```

Full list:

https://docs.python.org/3/library/unittest.html#test-cases

Test Methods And Assertions

A single test method will stop at the first failing assertion.

Group related assertions in one test method, and separate other groups into new methods.

```python
class TestSplitting(TestCase):
    def test_split_evenly(self):
        '''split_evenly() splits an integer into the smallest
           number of even groups.'''
        self.assertEqual([2, 2], split_evenly(4))
        self.assertEqual([5], split_evenly(5))
        self.assertEqual([6, 6], split_evenly(12))
        self.assertEqual([5, 5, 5], split_evenly(15))
    def test_split_amount(self):
        self.assertEqual([1], split_amount(1, 1))
        self.assertEqual([2, 2], split_amount(4, 2))
        self.assertEqual([2, 2, 1], split_amount(5, 3))
        self.assertEqual([3, 3, 2, 2, 2], split_amount(12, 5))
```

Test Methods and Failures

```
FF
======================================================================
FAIL: test_split_amount (test_splitting.TestSplitting)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_splitting.py", line 12, in test_split_amount
    self.assertEqual([1], split_amount(1, 1))
AssertionError: Lists differ: [1] != []

======================================================================
FAIL: test_split_evenly (test_splitting.TestSplitting)
split_evenly() splits an integer into the smallest # of even groups.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_splitting.py", line 7, in test_split_evenly
    self.assertEqual([2, 2], split_evenly(4))
AssertionError: Lists differ: [2, 2] != []

----------------------------------------------------------------------
Ran 2 tests in 0.001s
```

TDD

The idea of **Test-Driven Development**.

1. Write the test.
2. Run it, and watch it fail.
3. THEN write code to make the test pass.

This has some surprising benefits:

- Code clarity
- State of Flow
- Generally more robust software

And some downsides.

To TDD or Not?

People get religious about this. Be gentle with the zealots.

If you’re new to writing tests, strictly following TDD for a while is a great way to get very good, very quickly. And remember, writing good tests is a critical skill.

Once you’re fairly good at it: Consider following the 80-20 rule.

Lab: Unit Tests

In this self-directed lab, you implement a small library called `textlib`, and a test module named `test_textlib`.

Instructions: `testing/lab.txt`

- In labs/py3 for 3.x; labs/py2 for 2.7
- First follow the instructions to write `textlib.py` and `test_textlib.py`
- When you are done, give a thumbs up…
- … then follow the extra credit instructions

Remember, in Python 2, you MUST omit the `.py`:

```
python2.7 -m unittest test_textlib
```

In Python 3, you can pass the file name or the module name:

```
python3 -m unittest test_textlib.py
```

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".

```python
# 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

```python
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

```python
    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

```python
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("This is not a valid roman numeral.")
```

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("This is not a valid roman numeral.")
AssertionError: ValueError not raised

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

FAILED (failures=1)
```

Subtests

Python 3 only. And really valuable.

Imagine a function `numwords()`, counting unique words:

```
>>> numwords("Good, good morning. Beautiful morning!")
3
```

Testing numwords()

```
class TestWords(unittest.TestCase):
    def test_whitespace(self):
        self.assertEqual(2, numwords("foo bar"))
        self.assertEqual(2, numwords("    foo bar"))
        self.assertEqual(2, numwords("foo\tbar"))
        self.assertEqual(2, numwords("foo   bar"))
        self.assertEqual(2, numwords("foo bar    \t   \t"))
        # And so on, and so on...
```

This has two problems.

Less repetition...

```
    def test_whitespace_forloop(self):
        texts = [
            "foo bar",
            "     foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar     \t    \t",
            ]
        for text in texts:
            self.assertEqual(2, numwords(text))
```

More maintainble. But...

... but problematic

That approach creates more problems than it solves.

```
$ python3 -m unittest test_words_forloop.py
F
======================================================================
FAIL: test_whitespace_forloop (test_words_forloop.TestWords)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_forloop.py", line 17, in test_whitespace_forloop
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

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

FAILED (failures=1)
```

Pop quiz: what exactly went wrong?

A Better Way

We need something that is (a) maintainable, and (b) clear in the error reporting.

Python 3.4 solves this with **subtests**.

```
    def test_whitespace_subtest(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

self.subTest()

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

`self.subTest()` creates a context for assertions.

Even if that assertion fails, the test continues through the for loop.

ALL failures are collected and reported at the end, with clear information identifying the exact problem.

Subtest Reporting

```
$ python3 -m unittest test_words_subtest.py
=========================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo\tbar')
---------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

=========================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='foo bar    \t   \t')
---------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 4

---------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures=2)
```

Subtest Reporting

Behold the opulence of information in this output:

- Each individual failing input has its own detailed summary.
- We are told what the full value of `text` was.
- We are told what the actual returned value was, clearly compared to the expected value.
- No values are skipped. We can be confident that these two are the *only* failures.

In detail...

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

The key-value pairs to `subTest()` are used in reporting the output. They can be anything you like.

Pay attention: the symbol `text` has *two different meanings* on these lines.

- The argument to `numwords()`
- A field in the failure report

Reporting Fields

Suppose you wrote:

```
for text in texts:
    with self.subTest(input_text=text):
        self.assertEqual(2, numwords(text))
```

Then the failure output might look like:

```
FAIL: test_whitespace_subtest (test_words_subtest.TestWords)
(input_text='foo\tbar')
```

Lab: Intermediate Unit Tests

Instructions: `testing/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`
- When you are done, give a thumbs up.

Next, you can:

- Follow the extra credit instructions in `lab-intermediate.txt`.
- OR, if you're using Python 3, you can do `lab-subtests.txt`.

If you are new to `unittest`, focus on finishing `lab.txt` first.

Alternatives

`unittest` isn't the only game in town.

- doctest
  - Also in standard library
  - Most of your labs use this!
  - But only suitable for simpler code.
- pytest
  - Currently Python's most popular 3rd-party testing tool
  - Arguably better than unittest. But adds a separate dependency, and not universally used
- nose and nose2
  - Largely inactive now. Sometimes you'll still see it, though, especially with older projects.