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:

```python
# 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.

Lab: Simple Unit Tests

Let's practice. You'll write the smallest possible unit test, for a simple function called `greet()`.

Instructions: `lab-simple.txt`

- In labs/py3 for 3.x; labs/py2 for 2.7
- First follow the instructions to write `simple.py` and `test_simple.py`
- When you are done, give a thumbs up, and say HIGH FIVE! in the chat room.
- Then skim through **PythonicTDD.pdf**. Just notice what interests you.

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

```
python2.7 -m unittest test_simple
```

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

```
python3 -m unittest test_simple.py
```

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.

```
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.

```
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: `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
```

Alternatives

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

- doctest
  - Also in standard library
  - Labs in other Python courses use this!
  - But only suitable for simpler code.
- pytest
  - 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.