The Foundation: Automated Tests

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, including:

- **Unit test** - Tests a specific component (function, class, etc.) in isolation. Checks for a specific response to a particular input.
- **Integration test** - Checks that several components or sub-systems work together correctly. The *interactions* are being tested, more than the individual parts.

And others too.

We'll focus on the ideas common to all automated test types.

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

```
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
```

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.

Testing split_amount()

```
# 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))
```

```
$ 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.

Vocabulary

A **component** usually means either a function or a class.

It can also mean a module, or a tightly-knit group of several functions and classes. Some "primitive" unit of code you can write a unit test for.

The **system under test**, or SUT, means "whatever is being tested". It's more broad. The SUT can be:

- A single component
- A group of components
- An entire subsystem
- A microservice
- A distributed system of several applications

A **test suite** is a collection of related test cases. The "full suite of tests" means all the tests you have.

What's a "test"?

Does "running a test" mean:

- A single assertion?
- Executing a test method?
- Running all the test methods in a TestCase?
- Running a test suite?

Usually, "running a test" means "executing a single test method", and its assertions.

But this isn't strict at all. "Running a test" can mean any of the above.

"Running a test case" means running all the methods in a single `TestCase` subclass.

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` folder
- First follow the instructions to write `simple.py` and `test_simple.py`
- When you are done, study the solution - compare to what you wrote.

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
```