Testing Automatically Using unittest

doctest vs. unittest

Python's `unittest` module provides a testing framework that is similar to `doctest`.
Unlike `doctest`, which can make code hard to read when there are many tests, the `unittest` tests are written separately from the function being tested.

Translating `doctest` to `unittest`:

```
def get_divisors(num, possible_divisors):
    ''' (int, list of int) -> list of int

    Return a list of the values from possible_divisors
    that are divisors of num.

    >>> get_divisors(8, [1, 2, 3])
    [1, 2]
    >>> get_divisors(4, [-2, 0, 2])
    [-2, 2]
    '''

    divisors = []
    for item in possible_divisors:
        if item != 0 and num % item == 0:
            if num % item == 0:
                divisors.append(item)
    return divisors
```

→

```
import unittest
import divisors

class TestDivisors(unittest.TestCase):
    """Example unittest test methods for get_divisors."""

    def test_divisors_example_1(self):
        """Test get_divisors with 8 and [1, 2, 3]."""

        actual = divisors.get_divisors(8, [1, 2, 3])
        expected = [1, 2]
        self.assertEqual(expected, actual)

    def test_divisors_example_2(self):
        """Test get_divisors with 4 and [-2, 0, 2]."""

        actual = divisors.get_divisors(4, [-2, 0, 2])
        expected = [-2, 2]
        self.assertEqual(expected, actual)
```

Similarities and Differences

In doctest:

- `import doctest`
- Write the tests as you would type them in the shell.
- Write the expected result on the next line.

In unittest:

- `import unittest`
- Write separate methods for each test. In each method,
  - write a call on the function being tested, and
  - call `self.assertEqual(...)` to compare the *actual result* to the *expected result*.

To *assert* something is to claim that it is true.

Running Tests

To run tests using `doctest`:

- call `doctest.testmod()`, which examines the docstrings in the current module,
- executes the tests that it finds, and
- reports any differences between the actual results and the expected results.

To run tests using `unittest`:

- call `unittest.main()`, which examines all of the `TestCase` subclasses in the current module,
- calls each method that begin with "test", and
- and reports any unexpected results.

When calling `unittest` from within IDLE, the parameter `exit` should be assigned `False`:
`unittest.main(exit=False)`

Comparing Output

Successful Test

When we run the `test_divisors` module we get these results, where each dot represents a successful test:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.025s

OK
```

Test with Errors

Now, let's change the code as follows:

line 13: `divisors = []` → line 13: `divisors = [num]`

When the tests are executed, the following results are reported:

```
FF
======================================================================
FAIL: test_divisors_example_1 (__main__.TestDivisors)
Test get_divisors with 8 and [1, 2, 3].
----------------------------------------------------------------------
Traceback (most recent call last):
    File "test_divisors.py", line 13, in test_divisors_example_1
        self.assertEqual(actual, expected)
AssertionError: Lists differ: [8, 1, 2] != [1, 2]

First differing element 0:
8
1

First list contains 1 additional elements.
First extra element 2:
2

- [8, 1, 2]
?  ---

+ [1, 2]

======================================================================
FAIL: test_divisors_example_2 (__main__.TestDivisors)
Test get_divisors with 4 and [-2, 0, 2].
----------------------------------------------------------------------
Traceback (most recent call last):
    File "test_divisors.py", line 20, in test_divisors_example_2
            self.assertEqual(actual, expected)
AssertionError: Lists differ: [4, -2, 2] != [-2, 2]

First differing element 0:
4
-2

First list contains 1 additional elements.
First extra element 2:
2

- [4, -2, 2]
?  ---

+ [-2, 2]

----------------------------------------------------------------------
Ran 2 tests in 0.018s

FAILED (failures=2)
```

This time, we get a lot of feedback! We see:

- 2 Fs instead of 2 dots: [`FF`]
- the name of the method that has the failure: [`(FAIL: test_divisors_example_1(__main__.TestDivisors)`]
- the failed method's docstring: [`Test get_divisors with 8 and [1, 2, 3]`]
- a *traceback*, which is the series of function and method calls that led to the error: [`Traceback (most recent call last):...`]
- the `AssertionError`, including the expected and actual values: [`AssertionError: Lists differ: [8, 1, 2] != [1, 2]`]
- details about the problems: [`First differing element 0:...`]
- a summary of the results: [`FAILED (failures=2)`]

Let's fix that bug and introduce a different one:

```
                                  →
divisors = [num]                     divisors = []
for item in possible_divisors:       for item in possible_divisors:
```

```
    if item != 0 and num % item == 0:
        if num % item == 0:
            divisors.append(item)

return divisors
```

```
    # if item != 0 and num % item == 0:
    if num % item == 0:
        divisors.append(item)

return divisors
```

When the tests are executed, the following results are reported:

```
.E
======================================================================
ERROR: test_divisors_example_2 (__main__.TestDivisors)
Test get_divisors with 4 and [-2, 0, 2].
----------------------------------------------------------------------
Traceback (most recent call last):
    File "test_divisors.py", line 18, in test_divisors_example_2
        actual = divisors.get_divisors(4, [-2, 0, 2])
    File "divisors.py", line 16, in get_divisors
        if num % item == 0:
ZeroDivisionError: integer division or modulo by zero

----------------------------------------------------------------------
Ran 2 tests in 0.048s

FAILED (errors=1)
```

One test passed and a zero division error occured when the other test was executed. Instead of an `F`, which indicates an incorrect assertion, we see an `E`. The `E` indicates that a call on function `get_divisors()` resulted in an error.

Again, the results include the method name, docstring, and a traceback. This time the traceback shows us several steps:

- on line 18, `get_divisors` was called:

```
File "test_divisors.py", line 18, in test_divisors_example_2
    actual = divisors.get_divisors(4, [-2, 0, 2])
```

- on line 16, in function `get_divisors`, the code `num % item == 0:` results in a `ZeroDivisionError`:

```
File "divisors.py", line 16, in get_divisors
    if num % item == 0:
ZeroDivisionError: integer division or modulo by zero
```

By using `unittest` instead of `doctest`, we separate the testing of the function from the function definition, which allows us to write a lot of tests without affecting readability of the code.

Typically, we will write:

- one `TestCase` subclass for each function we want to test, and
- one test method for each function call.

Jennifer Campbell • Paul Gries
University of Toronto
