More Halting, More Fire

Along with testing numbers that are too large, you need to test numbers that are too small. As we noted in our functional requirements, Roman numerals cannot express 0 or negative numbers.

```
import roman2
print (roman2.to_roman(0))
#''

print (roman2.to_roman(-1))
#''
```

Well *that's* not good. Let's add tests for each of these conditions.

```
import unittest

class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 4000)  #①

    def test_zero(self):
        '''to_roman should fail with 0 input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)     #②

    def test_negative(self):
        '''to_roman should fail with negative input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)    #③
```

① The `test_too_large()` method has not changed since the previous step. I'm including it here to show where the new code fits.

② Here's a new test: the `test_zero()` method. Like the `test_too_large()` method, it tells the `assertRaises()` method defined in `unittest.TestCase` to call our `to_roman()` function with a parameter of 0, and check that it raises the appropriate exception, `OutOfRangeError`.

appropriate exception, `OutOfRangeError`.

③ The `test_negative()` method is almost identical, except it passes `-1` to the `to_roman()` function. If either of these new tests does *not* raise an `OutOfRangeError` (either because the function returns an actual value, or because it raises some other exception), the test is considered failed.

Now check that the tests fail:

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... FAIL

======================================================================
FAIL: to_roman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 86, in test_negative
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman

======================================================================
FAIL: to_roman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 82, in test_zero
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=2)
```

Excellent. Both tests failed, as expected. Now let's switch over to the code and see what we can do to make them pass.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 4000):                                          #①
        raise OutOfRangeError('number out of range (must be 1..3999)')  #②

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
```

```
    return result
```

① This is a nice Pythonic shortcut: multiple comparisons at once. This is equivalent to if not (`(0 < n) and (n < 4000)`), but it's much easier to read. This one line of code should catch inputs that are too large, negative, or zero.

② If you change your conditions, make sure to update your human-readable error strings to match. The `unittest` framework won't care, but it'll make it difficult to do manual debugging if your code is throwing incorrectly-described exceptions.

I could show you a whole series of unrelated examples to show that the multiple-comparisons-at-once shortcut works, but instead I'll just run the unit tests and prove it.

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.016s

OK
```