More Bad Input

Now that the `from_roman()` function works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in the `to_roman()` function, but you have a powerful tool at your disposal: regular expressions. (If you're not familiar with regular expressions, now would be a good time to read the regular expressions chapter.)

As you saw in Case Study: Roman Numerals, there are several simple rules for constructing a Roman numeral, using the letters `M`, `D`, `C`, `L`, `X`, `V`, and `I`. Let's review the rules:

- Sometimes characters are additive. `I` is `1`, `II` is `2`, and `III` is `3`. `VI` is 6 (literally, "5 and 1"), `VII` is `7`, and `VIII` is `8`.
- The tens characters (`I`, `X`, `C`, and `M`) can be repeated up to three times. At `4`, you need to subtract from the next highest fives character. You can't represent `4` as `IIII`; instead, it is represented as `IV` ("`1` less than `5`"). `40` is written as `XL` ("`10` less than `50`"), `41` as `XLI`, `42` as `XLII`, `43` as `XLIII`, and then `44` as `XLIV` ("`10` less than `50`, then `1` less than `5`").
- Sometimes characters are... the opposite of additive. By putting certain characters before others, you subtract from the final value. For example, at `9`, you need to subtract from the next highest tens character: `8` is `VIII`, but `9` is `IX` ("1 less than 10"), not `VIIII` (since the I character can not be repeated four times). `90` is `XC`, `900` is `CM`.
- The fives characters can not be repeated. `10` is always represented as `X`, never as `VV`. `100` is always `C`, never `LL`.
- Roman numerals are read left to right, so the order of characters matters very much. `DC` is `600`; `CD` is a completely different number (`400`, "`100` less than `500`"). `CI` is `101`; `IC` is not even a valid Roman numeral (because you can't subtract `1` directly from `100`; you would need to write

it as `XCIX`, "`10` less than `100`, then 1 less than `10`").

Thus, one useful test would be to ensure that the `from_roman()` function should fail when you pass it a string with too many repeated numerals. How many is "too many" depends on the numeral.

```
import unittest
class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Another useful test would be to check that certain patterns aren't repeated. For example, `IX` is `9`, but `IXIX` is never valid.

```
 def test_repeated_pairs(self):
        '''from_roman should fail with repeated pairs of numerals'''
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

A third test could check that numerals appear in the correct order, from highest to lowest value. For example, `CL` is `150`, but `LC` is never valid, because the numeral for `50` can never come before the numeral for `100`. This test includes a randomly chosen set of invalid antecedents: `I` before `M`, `V` before `X`, and so on.

```
 def test_malformed_antecedents(self):
        '''from_roman should fail with malformed antecedents'''
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Each of these tests relies the `from_roman()` function raising a new exception, `InvalidRomanNumeralError`, which we haven't defined yet.

```
# roman6.py
class InvalidRomanNumeralError(ValueError): pass
```

All three of these tests should fail, since the `from_roman()` function doesn't currently have any validity checking. (If they don't fail now, then what the heck are they testing?)

```
you@localhost:~/diveintopython3/examples$ python3 romantest6.py
FFF.......
======================================================================
FAIL: test_malformed_antecedents (__main__.FromRomanBadInput)
from_roman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 113, in test_malformed_antecedents
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_repeated_pairs (__main__.FromRomanBadInput)
from_roman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 107, in test_repeated_pairs
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_too_many_repeated_numerals (__main__.FromRomanBadInput)
from_roman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 102, in test_too_many_repeated_numerals
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 10 tests in 0.058s

FAILED (failures=3)
```

Good deal. Now, all we need to do is add the regular expression to test for valid Roman numerals into the `from_roman()` function.

```
import re
roman_numeral_pattern = re.compile('''
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    ''', re.VERBOSE)

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not roman_numeral_pattern.search(s):
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
```

```
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index : index + len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

And re-run the tests…

```
you@localhost:~/diveintopython3/examples$ python3 romantest7.py
..........
----------------------------------------------------------------------
Ran 10 tests in 0.066s

OK
```

And the anticlimax award of the year goes to… the word "`OK`", which is printed by the `unittest` module when all the tests pass.