A Pleasing Symmetry

Converting a string from a Roman numeral to an integer sounds more difficult than converting an integer to a Roman numeral. Certainly there is the issue of validation. It's easy to check if an integer is greater than 0, but a bit harder to check whether a string is a valid Roman numeral. But we already constructed a regular expression to check for Roman numerals, so that part is done.

That leaves the problem of converting the string itself. As we'll see in a minute, thanks to the rich data structure we defined to map individual Roman numerals to integer values, the nitty-gritty of the `from_roman()` function is as straightforward as the `to_roman()` function.

But first, the tests. We'll need a "known values" test to spot-check for accuracy. Our test suite already contains a mapping of known values; let's reuse that.

```
def test_from_roman_known_values(self):
        '''from_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

There's a pleasing symmetry here. The `to_roman()` and `from_roman()` functions are inverses of each other. The first converts integers to specially-formatted strings, the second converts specially-formated strings to integers. In theory, we should be able to "round-trip" a number by passing to the `to_roman()` function to get a string, then passing that string to the `from_roman()` function to get an integer, and end up with the same number.

```
n = from_roman(to_roman(n)) for all values of n
```

In this case, "all values" means any number between `1..3999`, since that is the valid range of inputs to the `to_roman()` function. We can express this symmetry in a test case that runs through all the values `1..3999` calls

synmetry in a test case that runs through all the values `1..3999`, calls

`to_roman()`, calls `from_roman()`, and checks that the output is the same as the original input.

```
import unittest

class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 4000):
            numeral = roman5.to_roman(integer)
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

These new tests won't even fail yet. We haven't defined a `from_roman()` function at all, so they'll just raise errors.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
E.E....
======================================================================
ERROR: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 78, in test_from_roman_known_values
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

======================================================================
ERROR: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 103, in test_roundtrip
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

----------------------------------------------------------------------
Ran 7 tests in 0.019s

FAILED (errors=2)
```

A quick stub function will solve that problem.

```
# roman5.py
def from_roman(s):
    '''convert Roman numeral to integer'''
```

(Hey, did you notice that? I defined a function with nothing but a docstring.

That's legal Python. In fact, some programmers swear by it. "Don't stub; document!")

Now the test cases will actually fail.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
F.F....
======================================================================
FAIL: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 79, in test_from_roman_known_values
    self.assertEqual(integer, result)
AssertionError: 1 != None

======================================================================
FAIL: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 104, in test_roundtrip
    self.assertEqual(integer, result)
AssertionError: 1 != None

----------------------------------------------------------------------
Ran 7 tests in 0.002s

FAILED (failures=2)
```

Now it's time to write the `from_roman()` function.

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:  #①
            result += integer
            index += len(numeral)
    return result
```

① The pattern here is the same as the to_roman() function. You iterate through your Roman numeral data structure (a tuple of tuples), but instead of matching the highest integer values as often as possible, you match the "highest" Roman numeral character strings as often as possible.

If you're not clear how `from_roman()` works, add a print statement to the end of the `while` loop:

```
def from_roman(s):
```

```
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print('found', numeral, 'of length', len(numeral), ', adding', integer)

import roman5
print (roman5.from_roman('MCMLXXII'))
#found M of length 1, adding 1000
#found CM of length 2, adding 900
#found L of length 1, adding 50
#found X of length 1, adding 10
#found X of length 1, adding 10
#found I of length 1, adding 1
#found I of length 1, adding 1
#1972
```

Time to re-run the tests.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
.......
----------------------------------------------------------------------
Ran 7 tests in 0.060s

OK
```

Two pieces of exciting news here. The first is that the `from_roman()` function works for good input, at least for all the known values. The second is that the "round trip" test also passed. Combined with the known values tests, you can be reasonably sure that both the `to_roman()` and `from_roman()` functions work properly for all possible good values. (This is not guaranteed; it is theoretically possible that `to_roman()` has a bug that produces the wrong Roman numeral for some particular set of inputs, and that `from_roman()` has a reciprocal bug that produces the same wrong integer values for exactly that set of Roman numerals that `to_roman()` generated incorrectly. Depending on your application and your requirements, this possibility may bother you; if so, write more comprehensive test cases until it doesn't bother you.)