A Single Question

A test case answers a single question about the code it is testing. A test case should be able to…

- …run completely by itself, without any human input. Unit testing is about automation.
- …determine by itself whether the function it is testing has passed or failed, without a human interpreting the results.
- …run in isolation, separate from any other test cases (even if they test the same functions). Each test case is an island.

Every test is an island.

Given that, let's build a test case for the first requirement:

1. The `to_roman()` function should return the Roman numeral representation for all integers `1` to `3999`.

It is not immediately obvious how this code does… well, *anything*. It defines a class which has no `__init__()` method. The class does have another method, but it is never called. The entire script has a `__main__` block, but it doesn't reference the class or its method. But it does do something, I promise.

```
import roman1
import unittest

class KnownValues(unittest.TestCase):               #①
    known_values = ( (1, 'I'),
                     (2, 'II'),
                     (3, 'III'),
                     (4, 'IV'),
                     (5, 'V'),
                     (6, 'VI'),
                     (7, 'VII'),
                     (8, 'VIII'),
                     (9, 'IX'),
                     (10, 'X'),
                     (10, 'X'),
                     (50, 'L'),
                     (100, 'C'),
                     (500, 'D'),
                     (1000, 'M'),
                     (31, 'XXXI'),
                     (148, 'CXLVIII'),
                     (294, 'CCXCIV'),
                     (312, 'CCCXII'),
                     (421, 'CDXXI'),
                     (528, 'DXXVIII'),
                     (621, 'DCXXI'),
                     (782, 'DCCLXXXII'),
                     (870, 'DCCCLXX'),
                     (941, 'CMXLI'),
                     (1043, 'MXLIII'),
                     (1110, 'MCX'),
                     (1226, 'MCCXXVI'),
                     (1301, 'MCCCI'),
                     (1485, 'MCDLXXXV'),
                     (1509, 'MDIX'),
                     (1607, 'MDCVII'),
                     (1754, 'MDCCLIV'),
                     (1832, 'MDCCCXXXII'),
                     (1993, 'MCMXCIII'),
                     (2074, 'MMLXXIV'),
                     (2152, 'MMCLII'),
                     (2212, 'MMCCXII'),
                     (2343, 'MMCCCXLIII'),
                     (2499, 'MMCDXCIX'),
                     (2574, 'MMDLXXIV'),
                     (2646, 'MMDCXLVI'),
                     (2723, 'MMDCCXXIII'),
                     (2892, 'MMDCCCXCII'),
                     (2975, 'MMCMLXXV'),
                     (3051, 'MMMLI'),
                     (3185, 'MMMCLXXXV'),
                     (3250, 'MMMCCL'),
                     (3313, 'MMMCCCXIII'),
                     (3408, 'MMMCDVIII'),
                     (3501, 'MMMDI'),
                     (3610, 'MMMDCX'),
                     (3743, 'MMMDCCXLIII'),
                     (3844, 'MMMDCCCXLIV'),
                     (3888, 'MMMDCCCLXXXVIII'),
                     (3940, 'MMMCMXL'),
                     (3999, 'MMMCMXCIX'))           #②

    def test_to_roman_known_values(self):           #③
        '''to_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman1.to_roman(integer)       #④
            self.assertEqual(numeral, result)       #⑤

if __name__ == '__main__':
    unittest.main()
```

① To write a test case, first subclass the `TestCase` class of the `unittest` module. This class provides many useful methods which you can use in your test case to test specific conditions.

② This is a tuple of integer/numeral pairs that I verified manually. It includes the lowest ten numbers, the highest number, every number that translates to a single-character Roman numeral, and a random sampling of other valid numbers. You don't need to test every possible input, but you should try to test all the obvious edge cases.

③ Every individual test is its own method. A test method takes no parameters, returns no value, and must have a name beginning with the four letters test. If a test method exits normally without raising an exception, the test is considered passed; if the method raises an exception, the test is considered failed.

④ Here you call the actual `to_roman()` function. (Well, the function hasn't been written yet, but once it is, this is the line that will call it.) Notice that you have now defined the api for the `to_roman()` function: it must take an integer (the number to convert) and return a string (the Roman numeral representation). If the api is different than that, this test is considered failed. Also notice that you are not trapping any exceptions when you call `to_roman()`. This is intentional. `to_roman()` shouldn't raise an exception when you call it with valid input, and these input values are all valid. If `to_roman()` raises an exception, this test is considered failed.

⑤ Assuming the `to_roman()` function was defined correctly, called correctly, completed successfully, and returned a value, the last step is to check whether it returned the *right* value. This is a common question, and the `TestCase` class provides a method, `assertEqual`, to check whether two values are equal. If the result returned from `to_roman()` (`result`) does not match the known value you were expecting (`numeral`), `assertEqual` will raise an exception and the test will fail. If the two values are equal, `assertEqual` will do nothing. If every value returned from `to_roman()` matches the known value you expect, `assertEqual` never raises an exception, so `test_to_roman_known_values` eventually exits normally, which means `to_roman()` has passed this test.

Write a test that fails, then code until it passes.

Once you have a test case, you can start coding the `to_roman()` function. First, you should stub it out as an empty function and make sure the tests fail. If the tests succeed before you've written any code, your tests aren't testing your code at all! Unit testing is a dance: tests lead, code follows. Write a test that fails, then code until it passes.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    pass                                       #①
```

① At this stage, you want to define the api of the `to_roman()` function, but you don't want to code it yet. (Your test needs to fail first.) To stub it out, use the Python reserved word `pass`, which does precisely nothing.

Execute `romantest1.py` on the command line to run the test. If you call it with the `-v` command-line option, it will give more verbose output so you can see exactly what's going on as each test case runs. With any luck, your output should look like this:

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)                       ①
to_roman should give known result with known input ... FAIL             ②

======================================================================
FAIL: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest1.py", line 73, in test_to_roman_known_values
    self.assertEqual(numeral, result)
AssertionError: 'I' != None                                             ③


----------------------------------------------------------------------
Ran 1 test in 0.016s                                                    ④

FAILED (failures=1)                                                     ⑤
```

① Running the script runs `unittest.main()`, which runs each test case. Each test case is a method within a class in `romantest.py`. There is no required organization of these test classes; they can each contain a single test method, or you can have one class that contains multiple test methods. The only requirement is that each test class must inherit from `unittest.TestCase`.

② For each test case, the `unittest` module will print out the `docstring` of the method and whether that test passed or failed. As expected, this test case fails.

③ For each failed test case, `unittest` displays the trace information showing exactly what happened. In this case, the call to `assertEqual()` raised an `AssertionError` because it was expecting `to_roman(1`) to return `'I'`, but it didn't. (Since there was no explicit return statement, the function returned `None`, the Python null value.)

④ After the detail of each test, `unittest` displays a summary of how many tests were performed and how long it took.

⑤ Overall, the test run failed because at least one test case did not pass. When a test case doesn't pass, `unittest` distinguishes between failures and errors. A failure is a call to an `assertXYZ` method, like `assertEqual` or `assertRaises`, that fails because the asserted condition is not true or the expected exception was not raised. An error is any other sort of exception raised in the code you're testing or the unit test case itself.

Now, finally, you can write the `to_roman()` function.

```
roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
                     ('L',  50),
                     ('XL', 40),
                     ('X',  10),
                     ('IX', 9),
                     ('V',  5),
                     ('IV', 4),
                     ('I',  1))                #①

def to_roman(n):
    '''convert integer to Roman numeral'''
    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:                    #②
            result += numeral
            n -= integer
    return result
```

① `roman_numeral_map` is a tuple of tuples which defines three things: the character representations of the most basic Roman numerals; the order of the Roman numerals (in descending value order, from `M` all the way down to `I`); the value of each Roman numeral. Each inner tuple is a pair of (numeral, value). It's not just single-character Roman numerals; it also defines two-character pairs like `CM` ("one hundred less than one thousand"). This makes the `to_roman()` function code simpler.

② Here’s where the rich data structure of `roman_numeral_map` pays off, because you don’t need any special logic to handle the subtraction rule. To convert to Roman numerals, simply iterate through `roman_numeral_map` looking for the largest integer value less than or equal to the input. Once found, add the Roman numeral representation to the end of the output, subtract the corresponding integer value from the input, lather, rinse, repeat.

If you’re still not clear how the `to_roman()` function works, add a `print()` call to the end of the `while` loop:

```
while n >= integer:
    result += numeral
    n -= integer
    print('subtracting {0} from input, adding {1} to output'.format(integer, numeral))
```

With the debug `print()` statements, the output looks like this:

```
import roman1
print (roman1.to_roman(1424))
#subtracting 1000 from input, adding M to output
#subtracting 400 from input, adding CD to output
#subtracting 10 from input, adding X to output
#subtracting 10 from input, adding X to output
#subtracting 4 from input, adding IV to output
#'MCDXXIV'
```

So the `to_roman(`) function appears to work, at least in this manual spot check. But will it pass the test case you wrote?

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok          ①

----------------------------------------------------------------------
Ran 1 test in 0.016s

OK
```

① Hooray! The `to_roman()` function passes the “known values” test case. It’s not comprehensive, but it does put the function through its paces with a variety of inputs, including inputs that produce every single-character Roman

numeral, the largest possible input (`3999`), and the input that produces the longest possible Roman numeral (`3888`). At this point, you can be reasonably confident that the function works for any good input value you could throw at it.

“Good” input? Hmm. What about bad input?