# Handling Changing Requirements

Despite your best efforts to pin your customers to the ground and extract exact requirements from them on pain of horrible nasty things involving scissors and hot wax, requirements will change. Most customers don't know what they want until they see it, and even if they do, they aren't that good at articulating what they want precisely enough to be useful. And even if they do, they'll want more in the next release anyway. So be prepared to update your test cases as requirements change.

Suppose, for instance, that you wanted to expand the range of the Roman numeral conversion functions. Normally, no character in a Roman numeral can be repeated more than three times in a row. But the Romans were willing to make an exception to that rule by having 4 `M` characters in a row to represent `4000`. If you make this change, you'll be able to expand the range of convertible numbers from `1..3999` to `1..4999`. But first, you need to make some changes to your test cases.

```
import unittest
class KnownValues(unittest.TestCase):
    known_values = ( (1, 'I'),
                     #.
                     #.
                     #.
                    (3999, 'MMMCMXCIX'),
                    (4000, 'MMMM'),                                #①
                    (4500, 'MMMMD'),
                    (4888, 'MMMMDCCCLXXXVIII'),
                    (4999, 'MMMMCMXCIX') )

class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''
        self.assertRaises(roman8.OutOfRangeError, roman8.to_roman, 5000)  #②

    #.
    #.
    #.

class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
```

```
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):     #③
            self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, s)

    #.
    #.
    #.

class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 5000):                                    #④
            numeral = roman8.to_roman(integer)
            result = roman8.from_roman(numeral)
            self.assertEqual(integer, result)
```

① The existing known values don't change (they're all still reasonable values to test), but you need to add a few more in the `4000` range. Here I've included `4000` (the shortest), `4500` (the second shortest), `4888` (the longest), and `4999` (the largest).

② The definition of "large input" has changed. This test used to call `to_roman()` with `4000` and expect an error; now that `4000-4999` are good values, you need to bump this up to `5000`.

③ The definition of "too many repeated numerals" has also changed. This test used to call `from_roman()` with `'MMMM'` and expect an error; now that `MMMM` is considered a valid Roman numeral, you need to bump this up to `'MMMMM'`.

④ The sanity check loops through every number in the range, from `1` to `3999`. Since the range has now expanded, this for loop need to be updated as well to go up to `4999`.

Now your test cases are up to date with the new requirements, but your code is not, so you expect several of the test cases to fail.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ERROR          ①
to_roman should give known result with known input ... ERROR            ②
from_roman(to_roman(n))==n for all n ... ERROR                          ③
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok
```

```
======================================================================
ERROR: from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 82, in test_from_roman_known_values
    result = roman9.from_roman(numeral)
  File "C:\home\diveintopython3\examples\roman9.py", line 60, in from_roman
    raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
roman9.InvalidRomanNumeralError: Invalid Roman numeral: MMMM

======================================================================
ERROR: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 76, in test_to_roman_known_values
    result = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)

======================================================================
ERROR: from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 131, in testSanity
    numeral = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)

----------------------------------------------------------------------
Ran 12 tests in 0.171s

FAILED (errors=3)
```

① The `from_roman()` known values test will fail as soon as it hits `'MMMM'`, because `from_roman()` still thinks this is an invalid Roman numeral.

② The `to_roman()` known values test will fail as soon as it hits `4000`, because `to_roman()` still thinks this is out of range.

③ The roundtrip check will also fail as soon as it hits `4000`, because `to_roman()` still thinks this is out of range.

Now that you have test cases that fail due to the new requirements, you can think about fixing the code to bring it in line with the test cases. (When you first start coding unit tests, it might feel strange that the code being tested is never “ahead” of the test cases. While it’s behind, you still have some work to do, and as soon as it catches up to the test cases, you stop coding. After you get used to it, you’ll wonder how you ever programmed without tests.)

```
import re
```

```
import re

roman_numeral_pattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 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 to_roman(n):
    '''convert integer to Roman numeral'''
    if not isinstance(n, int):
        raise NotIntegerError('non-integers can not be converted')
    if not (0 < n < 5000):                          #②
        raise OutOfRangeError('number out of range (must be 1..4999)')

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

def from_roman(s):
  pass
    #.
    #.
    #.
```

① You don't need to make any changes to the `from_roman()` function at all. The only change is to `roman_numeral_pattern`. If you look closely, you'll notice that I changed the maximum number of optional `M` characters from `3` to `4` in the first section of the regular expression. This will allow the Roman numeral equivalents of `4999` instead of `3999`. The actual `from_roman()` function is completely generic; it just looks for repeated Roman numeral characters and adds them up, without caring how many times they repeat. The only reason it didn't handle `'MMMM'` before is that you explicitly stopped it with the regular expression pattern matching.

② The `to_roman()` function only needs one small change, in the range check. Where you used to check `0 < n < 4000`, you now check `0 < n < 5000`. And you change the error message that you raise to reflect the new acceptable range (`1..4999` instead of `1..3999`). You don't need to make any changes to the rest of the function; it handles the new cases already. (It merrily adds `'M'` for each thousand that it finds; given `4000`, it will spit out `'MMMM'`. The only reason it didn't do this before is that you explicitly stopped it with the range

check.)

You may be skeptical that these two small changes are all that you need. Hey, don't take my word for it; see for yourself.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.203s

OK  ①
```

① All the test cases pass. Stop coding.

Comprehensive unit testing means never having to rely on a programmer who says "Trust me."