Diving In

Like it or not, bugs happen. Despite your best efforts to write comprehensive unit tests, bugs happen. What do I mean by “bug”? A bug is a test case you haven’t written yet.

```
import roman7
print (roman7.from_roman('')) #①
#0
```

① This is a bug. An empty string should raise an `InvalidRomanNumeralError` exception, just like any other sequence of characters that don’t represent a valid Roman numeral.

After reproducing the bug, and before fixing it, you should write a test case that fails, thus illustrating the bug.

```
import unittest

class FromRomanBadInput(unittest.TestCase):
    #.
    #.
    #.
    def testBlank(self):
        '''from_roman should fail with blank string'''
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, '') #①
```

① Pretty simple stuff here. Call `from_roman()` with an empty string and make sure it raises an `InvalidRomanNumeralError` exception. The hard part was finding the bug; now that you know about it, testing for it is the easy part.

Since your code has a bug, and you now have a test case that tests this bug, the test case will fail:

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... FAIL

from_roman should fail with malformed antecedents ... 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

======================================================================
FAIL: from_roman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest8.py", line 117, in test_blank
    self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, '')
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 11 tests in 0.171s

FAILED (failures=1)
```

Now you can fix the bug.

```
def from_roman(s):
    '''convert Roman numeral to integer'''
    if not s:                                                              #①
        raise InvalidRomanNumeralError('Input can not be blank')
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError('Invalid Roman numeral: {}'.format(s))  #②

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

① Only two lines of code are required: an explicit check for an empty string, and a `raise` statement.

② I don't think I've mentioned this yet anywhere in this book, so let this serve as your final lesson in string formatting. Starting in Python 3.1, you can skip the numbers when using positional indexes in a format specifier. That is, instead of using the format specifier `{0}` to refer to the first parameter to the `format()` method, you can simply use `{}` and Python will fill in the proper positional index for you. This works for any number of arguments; the first positional index for you. This works for any number of arguments, the first `{}` is `{0}`, the second `{}` is `{1}`, and so forth.

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... ok  ①
from_roman should fail with malformed antecedents ... 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 11 tests in 0.156s

OK  ②
```

① The blank string test case now passes, so the bug is fixed.

② All the other test cases still pass, which means that this bug fix didn't break anything else. Stop coding.

Coding this way does not make fixing bugs any easier. Simple bugs (like this one) require simple test cases; complex bugs will require complex test cases. In a testing-centric environment, it may *seem* like it takes longer to fix a bug, since you need to articulate in code exactly what the bug is (to write the test case), then fix the bug itself. Then if the test case doesn't pass right away, you need to figure out whether the fix was wrong, or whether the test case itself has a bug in it. However, in the long run, this back-and-forth between test code and code tested pays for itself, because it makes it more likely that bugs are fixed correctly the first time. Also, since you can easily re-run *all* the test cases along with your new one, you are much less likely to break old code when fixing new code. Today's unit test is tomorrow's regression test.