“Halt And Catch Fire”

It is not enough to test that functions succeed when given good input; you must also test that they fail when given bad input. And not just any sort of failure; they must fail in the way you expect.

```
import roman1
print (roman1.to_roman(4000))
#MMMM

print (roman1.to_roman(5000))
#MMMMM

print (roman1.to_roman(9000))  #①
#MMMMMMMMM
```

① That’s definitely not what you wanted — that’s not even a valid Roman numeral! In fact, each of these numbers is outside the range of acceptable input, but the function returns a bogus value anyway. Silently returning bad values is baaaaaaad; if a program is going to fail, it is far better if it fails quickly and noisily. “Halt and catch fire,” as the saying goes. The Pythonic way to halt and catch fire is to raise an exception.

The Pythonic way to halt and catch fire is to raise an exception.

The question to ask yourself is, “How can I express this as a testable requirement?” How’s this for starters:

```
The to_roman() function should raise an OutOfRangeError when given an inte
ger greater than 3999.
```

What would that test look like?

```
import unittest
import roman2

class ToRomanBadInput(unittest.TestCase):                                    #①
    def test_too_large(self):                                                #②
        '''to_roman should fail with large input'''
        self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)  #③
```

① Like the previous test case, you create a class that inherits from `unittest.TestCase`. You can have more than one test per class (as you'll see later in this chapter), but I chose to create a new class here because this test is something different than the last one. We'll keep all the good input tests together in one class, and all the bad input tests together in another.

② Like the previous test case, the test itself is a method of the class, with a name starting with `test`.

③ The `unittest.TestCase` class provides the `assertRaises` method, which takes the following arguments: the exception you're expecting, the function you're testing, and the arguments you're passing to that function. (If the function you're testing takes more than one argument, pass them all to `assertRaises`, in order, and it will pass them right along to the function you're testing.)

Pay close attention to this last line of code. Instead of calling `to_roman()` directly and manually checking that it raises a particular exception (by wrapping it in a try...except block), the `assertRaises` method has encapsulated all of that for us. All you do is tell it what exception you're expecting (`roman2.OutOfRangeError`), the function (`to_roman()`), and the function's arguments (`4000`). The `assertRaises` method takes care of calling `to_roman()` and checking that it raises `roman2.OutOfRangeError`.

Also note that you're passing the `to_roman()` function itself as an argument; you're not calling it, and you're not passing the name of it as a string. Have I mentioned recently how handy it is that everything in Python is an object?

So what happens when you run the test suite with this new test?

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ERROR                          ①
```

```
======================================================================
ERROR: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AttributeError: 'module' object has no attribute 'OutOfRangeError'      ②

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

① You should have expected this to fail (since you haven't written any code to pass it yet), but… it didn't actually "fail," it had an "error" instead. This is a subtle but important distinction. A unit test actually has three return values: pass, fail, and error. Pass, of course, means that the test passed — the code did what you expected. "Fail" is what the previous test case did (until you wrote code to make it pass) — it executed the code but the result was not what you expected. "Error" means that the code didn't even execute properly.

② Why didn't the code execute properly? The traceback tells all. The module you're testing doesn't have an exception called `OutOfRangeError`. Remember, you passed this exception to the `assertRaises()` method, because it's the exception you want the function to raise given an out-of-range input. But the exception doesn't exist, so the call to the `assertRaises()` method failed. It never got a chance to test the `to_roman()` function; it didn't get that far.

To solve this problem, you need to define the OutOfRangeError exception in roman2.py.

```
class OutOfRangeError(ValueError):  #①
    pass                            #②
```

① Exceptions are classes. An "out of range" error is a kind of value error — the argument value is out of its acceptable range. So this exception inherits from the built-in `ValueError` exception. This is not strictly necessary (it could just inherit from the base `Exception` class), but it feels right.

② Exceptions don't actually do anything, but you need at least one line of code to make a class. Calling `pass` does precisely nothing, but it's a line of Python code, so that makes it a class.

Now run the test suite again.

Now run the test suite again.

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... FAIL                          ①

======================================================================
FAIL: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AssertionError: OutOfRangeError not raised by to_roman                  ②

----------------------------------------------------------------------
Ran 2 tests in 0.016s

FAILED (failures=1)
```

① The new test is still not passing, but it's not returning an error either. Instead, the test is failing. That's progress! It means the call to the `assertRaises()` method succeeded this time, and the unit test framework actually tested the `to_roman()` function.

② Of course, the `to_roman()` function isn't raising the `OutOfRangeError` exception you just defined, because you haven't told it to do that yet. That's excellent news! It means this is a valid test case — it fails before you write the code to make it pass.

Now you can write the code to make this test pass.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if n > 3999:
        raise OutOfRangeError('number out of range (must be less than 4000)')  #①

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

① This is straightforward: if the given input (`n`) is greater than `3999`, raise an `OutOfRangeError` exception. The unit test does not check the human-readable string that accompanies the exception, although you could write another test that did check it (but watch out for internationalization issues for strings that

that did check it (but watch out for internationalization issues for strings that vary by the user's language or environment).

Does this make the test pass? Let's find out.

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok                          ①

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

① Hooray! Both tests pass. Because you worked iteratively, bouncing back and forth between testing and coding, you can be sure that the two lines of code you just wrote were the cause of that one test going from "fail" to "pass." That kind of confidence doesn't come cheap, but it will pay for itself over the lifetime of your code.