Refactoring

The best thing about comprehensive unit testing is not the feeling you get when all your test cases finally pass, or even the feeling you get when someone else blames you for breaking their code and you can actually prove that you didn't. The best thing about unit testing is that it gives you the freedom to refactor mercilessly.

Refactoring is the process of taking working code and making it work better. Usually, "better" means "faster", although it can also mean "using less memory", or "using less disk space", or simply "more elegantly". Whatever it means to you, to your project, in your environment, refactoring is important to the long-term health of any program.

Here, "better" means both "faster" and "easier to maintain." Specifically, the `from_roman()` function is slower and more complex than I'd like, because of that big nasty regular expression that you use to validate Roman numerals. Now, you might think, "Sure, the regular expression is big and hairy, but how else am I supposed to validate that an arbitrary string is a valid a Roman numeral?"

Answer: there's only 5000 of them; why don't you just build a lookup table? This idea gets even better when you realize that *you don't need to use regular expressions at all*. As you build the lookup table for converting integers to Roman numerals, you can build the reverse lookup table to convert Roman numerals to integers. By the time you need to check whether an arbitrary string is a valid Roman numeral, you will have collected all the valid Roman numerals. "Validating" is reduced to a single dictionary lookup.

And best of all, you already have a complete set of unit tests. You can change over half the code in the module, but the unit tests will stay the same. That means you can prove — to yourself and to others — that the new code works just as well as the original.

```
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
class InvalidRomanNumeralError(ValueError): pass

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))

to_roman_table = [ None ]
from_roman_table = {}

def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]

def build_lookup_tables():
    def to_roman(n):
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)
        to_roman_table.append(roman_numeral)
        from_roman_table[roman_numeral] = integer

build_lookup_tables()
```

Let’s break that down into digestable pieces. Arguably, the most important line is the last one:

```
build_lookup_tables()
```

You will note that is a function call, but there’s no `if` statement around it. This is not an if `__name__ == '__main__'` block; it gets called *when the module is imported*. (It is important to understand that modules are only imported once, then cached. If you import an already-imported module, it does nothing. So this code will only get called the first time you import this module.)

So what does the `build_lookup_tables()` function do? I’m glad you asked.

```
to_roman_table = [ None ]
from_roman_table = {}
#.
#.
#.
def build_lookup_tables():
    def to_roman(n):                                #①
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)           #②
        to_roman_table.append(roman_numeral)        #③
        from_roman_table[roman_numeral] = integer
```

① This is a clever bit of programming… perhaps too clever. The `to_roman()` function is defined above; it looks up values in the lookup table and returns them. But the `build_lookup_tables()` function redefines the `to_roman()` function to actually do work (like the previous examples did, before you added a lookup table). Within the `build_lookup_tables()` function, calling `to_roman()` will call this redefined version. Once the `build_lookup_tables()` function exits, the redefined version disappears — it is only defined in the local scope of the `build_lookup_tables()` function.

② This line of code will call the redefined `to_roman()` function, which actually calculates the Roman numeral.

③ Once you have the result (from the redefined `to_roman()` function), you add the integer and its Roman numeral equivalent to both lookup tables.

Once the lookup tables are built, the rest of the code is both easy and fast.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]                                         #①

def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]                                       #②
```

① After doing the same bounds checking as before, the `to_roman()` function simply finds the appropriate value in the lookup table and returns it.

② Similarly, the `from_roman()` function is reduced to some bounds checking and one line of code. No more regular expressions. No more looping. `O(1)` conversion to and from Roman numerals.

But does it work? Why yes, yes it does. And I can prove it.

```
you@localhost:~/diveintopython3/examples$ python3 romantest10.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 0.031s                                               ①

OK
```

① Not that you asked, but it’s fast, too! Like, almost 10× as fast. Of course, it’s not entirely a fair comparison, because this version takes longer to import (when it builds the lookup tables). But since the import is only done once, the startup cost is amortized over all the calls to the `to_roman()` and `from_roman()` functions. Since the tests make several thousand function calls (the roundtrip test alone makes 10,000), this savings adds up in a hurry!

The moral of the story?

- Simplicity is a virtue.
- Especially when regular expressions are involved.
- Unit tests can give you the confidence to do large-scale refactoring.