(Not) Diving In

Kids today. So spoiled by these fast computers and fancy "dynamic" languages. Write first, ship second, debug third (if ever). In my day, we had discipline. Discipline, I say! We had to write programs by *hand*, on *paper*, and feed them to the computer on *punchcards*. And we *liked* it!

In this chapter, you're going to write and debug a set of utility functions to convert to and from Roman numerals. You saw the mechanics of constructing and validating Roman numerals in "Case study: roman numerals". Now step back and consider what it would take to expand that into a two-way utility.

The rules for Roman numerals lead to a number of interesting observations:

1. There is only one correct way to represent a particular number as a Roman numeral.
2. The converse is also true: if a string of characters is a valid Roman numeral, it represents only one number (that is, it can only be interpreted one way).
3. There is a limited range of numbers that can be expressed as Roman numerals, specifically `1` through `3999`. The Romans did have several ways of expressing larger numbers, for instance by having a bar over a numeral to represent that its normal value should be multiplied by `1000`. For the purposes of this chapter, let's stipulate that Roman numerals go from `1` to `3999`.
4. There is no way to represent `0` in Roman numerals.
5. There is no way to represent negative numbers in Roman numerals.
6. There is no way to represent fractions or non-integer numbers in Roman numerals.

Let's start mapping out what a `roman.py` module should do. It will have two main functions, `to_roman()` and `from_roman()`. The `to_roman()` function should take an integer from `1` to `3999` and return the Roman numeral representation as a string…

Stop right there. Now let's do something a little unexpected: write a test case that checks whether the `to_roman()` function does what you want it to. You read that right: you're going to write code that tests code that you haven't written yet.

This is called *test-driven development*, or TDD. The set of two conversion functions — `to_roman()`, and later `from_roman()` — can be written and tested as a unit, separate from any larger program that imports them. Python has a framework for unit testing, the appropriately-named `unittest` module.

Unit testing is an important part of an overall testing-centric development strategy. If you write unit tests, it is important to write them early and to keep them updated as code and requirements change. Many people advocate writing tests before they write the code they're testing, and that's the style I'm going to demonstrate in this chapter. But unit tests are beneficial no matter when you write them.

- Before writing code, writing unit tests forces you to detail your requirements in a useful fashion.
- While writing code, unit tests keep you from over-coding. When all the test cases pass, the function is complete.
- When refactoring code, they can help prove that the new version behaves the same way as the old version.
- When maintaining code, having tests will help you cover your ass when someone comes screaming that your latest change broke their old code. ("But sir, all the unit tests passed when I checked it in…")
- When writing code in a team, having a comprehensive test suite dramatically decreases the chances that your code will break someone else's code, because you can run their unit tests first. (I've seen this sort of thing in code sprints. A team breaks up the assignment, everybody takes the specs for their task, writes unit tests for it, then shares their unit tests with the rest of the team. That way, nobody goes off too far into developing code that doesn't play well with others.)