Extracted from:

Data Crunching

Solve Everyday Problems Using Java, Python, and More

Chapter 7

Horseshoe Nails

There's an old children's rhyme that goes

> For want of a nail the shoe was lost.
> For want of a shoe the horse was lost.
> For want of a horse the rider was lost.
> For want of a rider the battle was lost.
> For want of a battle the kingdom was lost.
> And all for the want of a horseshoe nail.

In honor of that rhyme, engineers sometimes use the term *horseshoe nails* to refer to those apparently trivial things that can bring the whole system crashing down when they go wrong.

This chapter discusses some of the horseshoe nails of data crunching. No matter what tools you're crunching data with, these topics come up again and again.

7.1 Unit Testing

When historians of the future come to write the annals of our times, they—sorry, that was a bit pretentious. Let me try again....

In retrospect, the real revolution in programming in the last decade hasn't been the shift to distributed net-based systems. Instead, it has been the spread of test-driven development. Whether you buy into agile methodologies like Extreme Programming or not, you can't deny that

- tools like the JUnit testing framework have made testing easier than ever before;
- as a result, more programmers are writing and running tests on a daily (or even hourly) basis;

- the software they're creating is thereby more reliable than it would otherwise have been; and
- the software they're creating is also cleaner, since unit testing forces programmers to modularize more and makes refactoring easier and safer.

It's tempting to skip testing when crunching data, especially if the crunching in question has to be done only once (e.g., to reformat a legacy data file). However, experience has taught me that testing doesn't just find errors in my programs: it also keeps me honest. If I know that I'm going to write a few tests for a piece of code, I'm more likely to write the code itself cleanly and correctly.

That said, I usually test data crunching programs less rigorously than production code. When I'm working on the latter, I write unit tests to make sure that each method does what it's supposed to. When this proves difficult, I refactor my code to make testing easier, since something that's hard to test will almost always be hard to upgrade or replace in future.

When I'm writing a data crunching program, on the other hand, I usually concentrate on end-to-end functional testing; in other words, I feed it some data and check the results. I only start writing separate unit tests for the input, processing, and output when things go so badly wrong that I can't fix them in five minutes or less. Some of my colleagues[1] believe that this is terribly misguided and that I'm putting my soul in jeopardy by not always writing unit tests up front, but it seems to work for me. (That said, the older I get, the more preemptive testing I do....)

The first step in testing a data crunching program is deciding how you're going to do it. Broadly speaking, the two options are sampling and auditing, or comparing your program's output to examples you know are correct. If you choose the latter, you can do the comparison externally using tools like diff, or use string I/O to construct self-contained test suites for frameworks like JUnit. The following three sections look at each option in turn.

Sampling and Auditing

During rush hour in Toronto, I can get on the street car by either the front or the rear doors. I'm supposed to board at the rear only if I have a monthly pass or a transfer from the subway or a bus. No one is there to check—it would be too expensive to put two conductors on every car. Every once in a while, though, inspectors board the trolleys and check that everyone has some proof of payment. Anyone who doesn't is fined on the spot; penalties are steep enough that almost everyone buys a ticket rather than take the risk.

[1] Including the editors of this series.

Purists sometimes turn their noses up at random sampling, since it isn't guaranteed to catch mistakes, but sampling is often the most cost-effective way to find out if a data crunching program is working correctly. For example, suppose that you're processing expense claims from customer support engineers. Your input is three files. The first holds information about employees:

```
# NAME          EMAIL           ID
Hunt, Dave      dave            3001
Thomas, Andy    andy            3002
Wilson, Greg    gvwilson        4001
```

The second holds information about their expenses:

```
# ID         DATE         CURRENCY  AMOUNT  REASON
gvwilson     2004-11-18   CDN       48.22   books
andy         2004-12-09   US        149.95  printer/scanner
andy         2004-12-10   US        79.95   toner
andy         2004-12-20   US        79.95   toner
gvwilson     2004-12-25   CDN       127.05  phone charges
andy         2005-01-03   US        79.95   toner
```

and the third is a currency conversion table:

```
# FROM          TO              RATE
CDN             US              0.7954
```

You're supposed to produce a file with a running total of each employee's expenses, in American dollars, sorted by employee name and date:

```
# NAME          DATE            TOTAL
Wilson, Greg    2004-11-18      38.35
                2004-12-25      139.40
Thomas, Andy    2004-12-09      149.95
                2004-12-10      229.90
                2004-12-20      308.85
                2005-01-03      389.80
```

In order to make sure that the little Python program you wrote is working, you could choose twenty lines of input at random, run it through your program, and check the output by hand. If you do this, it's important that you save the input, or have some way to reselect exactly the same values, so that if you find any problems, you can rerun your program on exactly the same test data in order to check your fixes. The reason is that if you select a different input set for checking, it may not tickle the original error, so you may not be able to tell whether your fix is correct.

You should also select your test input randomly, rather than using a regular rule like "every 100^{th} line." Data often contains patterns; if your tests happen to resonate with those patterns, they can skip over every input record of a certain kind and leave some pretty nasty bugs in your program.

This may sound far-fetched, but it comes up surprisingly often in practice. For example, I once had to reformat some data from a set of experiments that mea-

sured how well firefighters' helmets withstood impact shocks (like falling bricks) after varying degrees of thermal degradation (what you and I would call melting). In order to test my program, I put every $hundred^{th}$ input record in a separate file and then ran my cruncher on that and drew a graph. Everything looked fine—at first.

What I didn't know was that the experiments had been done in fixed-size batches: ten different weights were dropped on each helmet after it had been been heated to one of ten different temperatures. As a result, I was sampling only the data for test #1 on each helmet. This meant that all of my test data had been entered by the same guy. Unfortunately, the other two technicians working on the problem entered their data differently....

So, how can you select random input? The simplest way is to read the input as normal and then use the ratio between the number of records you have and the number you want as the probability of keeping any particular record. You can then use your favorite random number generator to decide which records to keep. For example, here's a function that selects approximately N lines from a list at random:

```
def keepN(lines, N):
    # Make sure random number generation is reproducible.
    random.seed(12345213)
    # If not enough data, return what's there.
    if (not lines) or (len(lines) <= N):
        return lines[:]
    # Probability of keeping any line.
    prob = float(N) / len(lines)
    # Select.
    result = []
    for l in lines:
        if random.random() < prob:
            result.append(l)
    return result
```

This is also another good argument in favor of writing data crunching programs as a series of filter functions: the random selector is just another filter that may or may not be run after reading the initial data.

Diff

Checking a program's output by hand over and over again is so tedious and error-prone that most of us would rather skip testing than do it. One way to ease the tedium is to automate the checking using diff, a tool that prints out the differences between two text files.[2]

[2]You can also use Python's difflib library or similar libraries in other languages.

For instance, if `a.txt` contains

```
one
two
three
four
```

and `b.txt` contains

```
two
three
three
five
```

then `diff a.txt b.txt` prints out

```
1d0
< one
4c3,4
< four
---
> three
> five
```

The alphanumeric codes `1d0` and `4c3,4` are actually commands for an editor that hasn't been used by anyone except die-hard Unix nerds in twenty-five years. Each introduces a section showing a difference between the two files: < indicates lines from the first one, and > indicates lines from the second.

So, suppose you know what output your data crunching program is supposed to produce for some test input data. If you put that output in a file called expected.txt, you can run your cruncher and compare its actual output to what you expected in a single step, like this:

```
crunch < input | diff - expected.txt
```

The - argument to diff makes it read one set of data from standard input, rather than a file. If crunch's output is identical to what's in expected.txt, the command above won't print anything at all. If there are differences, on the other hand, diff will show you where they are. You can then patch your cruncher and rerun your test.[3]

If your shell has a "repeat previous command" feature (and they all do these days), that's enough—until you're dragged away to put out another fire and can't get back to processing expense claims for a week. If you had built up a dozen or more test cases, the odds are pretty long that you won't remember all of them.

For this reason, whenever I'm doing anything that will take more than five minutes to complete, I (almost) always start by writing a little Makefile. This Makefile doesn't recompile anything (unless I'm using Java or some other sturdy language).

[3] If diff's output is too hard to read, you can use a GUI tool such as windiff, xdiff, or meld.

Make

Make is one of the most widely used programming tools in the world. It was invented by Stuart Feldman at Bell Labs; he noticed that everyone wrote the same kind of shell script over and over again to recompile their programs, and decided that a specialized tool would make everyone's life easier. (Feldman went on to become a vice president of IBM, which shows you how far a good tool can take you.)

Like many classic Unix tools, Make's configuration files (called Makefiles) are written in an idiosyncratic syntax. As Make has grown more complex over the years, so too have Makefiles: real ones now contain function calls, conditionals, and many other features.

A Makefile contains zero or more rules, each of which has a head and a body. The head specifies a dependency between one or more *targets* and zero or more *prerequisites*. Targets and prerequisites are typically filenames, such as object files, HTML pages, and so on. Targets and prerequisites are separated from one another by spaces, and the set of targets is separated from the prerequisites by a single colon.

The body specifies zero or more *actions*, which are shell commands for bringing the target up to date. If any of the rule's prerequisites are newer than the target, Make executes the actions. In this example, the program xyz is built from the source file xyz.c, and the header file defs.h.

```
xyz : xyz.c defs.h
        cc -o xyz xyz.c
```

Here, xyz is the target, xyz.c and defs.h are the prerequisites, and the body contains a single action. A more complex example is

```
# Build the program
xyz xyz.lst : xyz.o util.o
        cc -WZall -o xyz xyz.o util.o

# Build the objects
xyz.o : xyz.c defs.h
        cc -c xyz.c

util.o : util.c defs.h
        cc -c util.c

# Clean up
clean :
        rm -f xyz xyz.lst *.o *~
```

The most important feature of Make is that it keeps track of dependencies *between* rules: if util.c changes, Make recompiles util.o, notices that util.o is now newer than xyz and xyz.lst, and rebuilds those two files. Similarly, if you tell Make that certain tests depend on certain files, then when those files change, Make will automatically rerun exactly the right set of tests.

Instead, each of its rules reruns one of my tests, while another rule runs *all* of my tests, in order. An example is

```
all : t_empty t_single t_dup t_long t_name t_currency
t_empty :
        python crunch.py < empty-input.txt | diff - empty-output.txt
t_single :
        python crunch.py < single-input.txt | diff - single-output.txt
          :         :          :
```

If I type make, Make runs each of my tests for me. If everything is working as it should, all I'll see is the commands echoed to the screen one after another. If I see any diff output, I know that something has gone wrong, and I can go and fix it.

String I/O

If I'm going to go to the trouble of putting tests in a Makefile, I might as well go one step further and write proper unit tests. In fact, if I'm serious about having my data crunching program do the right thing, I should write those tests *before* I start writing the cruncher. This forces me to think through what I'm supposed to do to the data, rather than retrofitting my expectations around the code I've already written. It also gives me a concrete goal to work toward: as soon as all my tests run, I can stop fiddling with the program and go on to my next task.

These days, the standard way to write unit tests is to use a framework like JUnit (for Java) or one of its clones like unittest (for Python), NUnit (for .NET), or Test::Unit (for Ruby). Each framework has its own quirks but typically works as follows:

- Derive a class from a base class provided by the framework (in Java, the base class is junit.framework.TestCase).
- Add one method to that class for each test you want to run. These methods must follow a few simple rules—for example, they must take no arguments, they must return nothing, and their names must start with the letters *test*—but other than that, they can do whatever they want.
- Inside each method, the programmer creates data structures, calls functions, and so on, and then checks the results using special assertion methods provided by the framework. If the assertions pass, the framework adds one to the count of successful tests. If any of the assertions fail, or if something goes wrong inside the test itself, the framework catches and reports the error.

This pattern works well when the tests are self-contained, but data crunching programs almost always need external data as input and are expected to produce

Test-Driven Development

One of the core practices of Extreme Programming is *test-driven development* (TDD), which tells programmers to write their test cases *before* they write their code. While this seems backward, it has several benefits:

- It forces you to think about exactly what your code is going to do, before you write it.
- It gives you goalposts to aim for. When all the tests pass, you're done.
- It keeps you honest by making it harder to change the problem definition (consciously or unconsciously) in order to avoid difficult coding.
- It means you actually *do* write tests.

TDD is a great way to approach any data crunching problem that can't be solved with a simple command-line pipe. Pick a few samples of input (or make them up), figure out what the output should be, stick them side by side in a Makefile or some other testing harness, and then write your cruncher. I promise, it'll be faster than writing the cruncher and then tweaking it over and over again to eliminate bugs.

(For more on test-driven development, see (Bec02).)

external data (such as new files) as output. You *can* put the input and expected output data in files and have your unit tests read and diff them, but there's a better way.

The trick is that almost every language in use today has some way of treating a string as a file. In C++, the class that does this is called `stringstream`; in Python, it's `StringIO` (and its faster cousin, `cStringIO`). Java provides a pair of classes called `StringReader` and `StringWriter`, .NET gives you `MemoryStream`, and so on. In each case, the string I/O class can be used in place of a "real" file, causing the program to read from, or write to, a buffer in memory. Here's a simple example in Python:

```
import cStringIO

data = """This is
a multi-line string
but we will read it
as if it were
a file."""
input = cStringIO.StringIO(data)
for line in input:
    print len(line)
input.close()
```