Extracted from:

Data Crunching

Solve Everyday Problems Using Java, Python, and More

Chapter 1

Introduction

A friend of mine used to have a sign over his desk that said, "The first 90% of the work takes the first 90% of the time. The other 10% of the work takes the other 90% of the time." If you're a programmer, some of that "other 90%" is probably spent crunching data. This book will show you how to do it faster and how to get it right on the first try.

So what exactly is *data crunching*? The easiest way to explain is with a couple of stories....

1.1 Name That Molecule

A few years ago, when 3D graphics cards were still a novelty on PCs, a high school science teacher asked me to help her create images of various molecules. In one hand, she had some files that specified the coordinates of the atoms in water, ethanol, caffeine, and other compounds. Each file was in PDB (Protein Data Bank) format and looked something like this:

```
COMPND      Ammonia
AUTHOR      DAVE WOODCOCK  97 10 31
ATOM      1  N           1       0.257  -0.363   0.000
ATOM      2  H           1       0.257   0.727   0.000
ATOM      3  H           1       0.771  -0.727   0.890
ATOM      4  H           1       0.771  -0.727  -0.890
TER       5              1
END
```

In the other hand, she had a Fortran program that could draw spheres. Its input had to be in a format called VU3, which looked like this:

```
-- Nitrogen
1 17 0.5 0.257 -0.363 0.000
-- Hydrogens
1 6 0.2 0.257 0.727 0.000
1 6 0.2 0.771 -0.727 0.890
1 6 0.2 0.771 -0.727 -0.890
```

Each line represents a single object. "1" means "sphere"; "17" is the color code for purple, "6" for gray, and the other numbers are the sphere's radius and XYZ coordinates.

Translating the file for ammonia from PDB to VU3 took about thirty seconds using Notepad. At that rate, translating the file for cholesterol (which has 78 atoms) would have taken several minutes; fixing the typos that would inevitably creep in would probably have taken a few minutes more, and we would still have the other 100-plus molecules to deal with.

Data crunching to the rescue. What we needed was a program that would do the translations for us. We didn't really care how fast it ran, but it had to be easy to write (or else it would have been more economical for my friend to assign the translation of the molecule data to her students as homework).

It took me about three minutes to write a program to do the job and another couple of minutes to find and fix two bugs in it. Translating 112 molecules then took less than a minute. A month later, when my friend switched graphics programs, it took her only a couple of minutes to modify the code to generate the files the new program needed. Ten minutes of programming saved her and her students several hours of tedious typing.

1.2 There's One in Every Crowd...

A few years later, I was teaching a programming class at the University of Toronto. Five graduate students had been hired to mark my students' first assignment. Four of them submitted their grades via a web form, which created a file that looked like this:

```
<marks marker="Aliyah K." date="2003-09-27"
       course="csc207" exer="e1">
  <mark sid="g3allanj" grade="7.5"></mark>
  <mark sid="g3kowrem" grade="9.0"></mark>
  <mark sid="g3daniel" grade="2.0">Incomplete (no Makefile)</mark>
  ...
</marks>
```

The fifth marker—well, he must have had more important things to do than fill in HTML forms, because what I got from him was an email containing this:

```
g3andyh         6/10
g3davet         4/10    # More comments than code
g4mclark        2.5     # Infinite loop in part 3
...
```

Never mind the fact that it wasn't in XML—the grades themselves were written in two different ways. There were typos, too: all of the students' IDs should have started with *g3*, but he had some *g4*s and *f3*s in there as well.

When I bounced his message back to him, I got an autoreply saying that he was out of town for a conference and wouldn't be back for two weeks (which was long after marks were due). After adding a note to my calendar to chew him out, I saved his message to a file, edited it to strip off the mail headers, and started writing code. In less than two minutes, I was reading in his data; a minute after that, I had a list of all the student IDs in the file that *weren't* in the class list. It took a minute of editing to fix them and another minute to double-check my program's output, and I was done. It was five minutes I shouldn't have had to spend, but it was still a lot faster than anything else I could have done.

1.3 And the Moral Is...

On the face of it, Stone Age Fortran graphics and XML grade files don't have much in common. In both cases, though, a few simple techniques let me do things that would otherwise have been too time-consuming to be practical. All over the world, programmers use those same techniques every day to recycle legacy data, translate from one vendor's proprietary format into another's, check configuration files, and search through web logs to see how many people have downloaded the latest release of their product.

This kind of programming is usually called *data crunching* or *data munging*. These terms cover the tasks described above and dozens of others that are equally unglamorous but just as crucial.

There's no "grand unified theory" of data crunching, but a few fundamental patterns do crop up over and over again, regardless of what language the solution is written in or the exact details of the problem being solved. In this book, I'll show you what those patterns are, when you should use them, and how they'll make your life easier. Along the way, I'll introduce you to some handy, but underused, features of Java, Python, and other languages. I'll also show you how to test data crunching programs so that when it's *your* turn to reformat a file full of grades, you won't accidentally give someone a zero they don't deserve.

1.4 Questions About Data Crunching

Before diving into those patterns, I should probably answer three questions.

What Do You Need to Get Started?

In order to show you how to crunch data, this book has to show you examples, which have to be written in particular languages. I hope every example will be readable even if you haven't seen the language it's written in before (if it isn't, please let me know). If you want to try them out yourself, you'll need the following:

- A handful of classic Unix command-line tools. If you use Linux, Solaris, Mac OS X, or one of their cousins, you already have these. On Windows, you can install Cygwin, a collection of free-as-in-free software that is available at `http://www.cygwin.com`. Cygwin includes dozens of packages; you need only the defaults for this book, but you may find many of the others useful.
- Python. This is included by default in most Linux installations and in Mac OS X since release 10.3 (Panther); installers for other platforms are available at `http://www.python.org`. If you're using Windows, you might prefer the ActiveState installer, from `http://www.activestate.com`, which includes some useful Windows-specific extensions. I used Python 2.3 when writing this book, but any version from 2.2 onward will work just as well.
- Java. I believe that agile languages such as Python, Perl, and Ruby are the best way to tackle most data crunching problems. Since you may want to use a sturdy language, such as Java, C++, or C#,[1] I've included examples in Java to show you that the same ideas work with them just as well.
- A command-line XSLT processor. I used xsltproc, which is available in most Linux distributions and as part of Cygwin's libxml2 and libxslt packages.
- A relational database with a command-line interface. MySQL and PostgreSQL are the best-known names in the open source world, but I'm very fond of SQLite,[2] a small, lightweight system designed to be embedded directly into other programs. It isn't as fast as its bigger brothers and doesn't have nearly as many features, but setup is trivial, and its error messages are surprisingly helpful.
- Source code for the examples. You can download the source from the book's home page at `www.pragmaticprogrammer.com/titles/gwd`.

[1] For example, you might not be allowed to add another tool to the build environment, or it might be simpler to write your cruncher in C# than to figure out how to call a Python script from .NET.

[2] `http://www.sqlite.org`

Joe Asks...

What About GUIs?

One thing you *won't* find in this book is any discussion of graphical interfaces. The reason is it's hard for other programs to read what GUIs produce. As you'll see over and over again in this book, data crunching tools are more effective in combination than they are on their own.

That said, you can do a lot of sophisticated data crunching using GUI tools such as Microsoft Excel and OpenOffice Calc. Programmers (particularly Linux geeks) may sneer, but many accountants can do more in five minutes with Excel, Microsoft Access, and a little VB than most programmers can do in half an hour with their favorite programming language. A GUI is also a must when it comes time to display data to people—ASCII bar charts just don't cut it in the Twenty-First Century.

Is "Data Crunching" Just a Fancy Name for "Quick-and-Dirty Hacking"?

No. The fastest way to produce a working solution is always to program well. Even if you expect to run a piece of code only once, you should write meaningful comments, use sensible variable names, run a few tests, and do all of the other things Dave and Andy told you to do in [HT00]. Why? Because if you don't make a habit of doing those things all the time, every time, you won't do them the one time they would have saved you three hours of debugging, two hours before a deadline.

When Should I Use These Techniques?

The answer is simple: use these techniques whenever they'll save you time *in the long run*. If you want to copy ten numbers from one web site to another, the best thing to do is to open a second browser and start cutting and pasting. At the other end of the scale, if you have to move eighty million purchase records spread across eleven tables from an Oracle database into the CRM package your company just bought, then you have a full-blown software development project on your hands. The odds are pretty good that if you sit down and do some analysis and design, you'll have a working solution faster. What's more, whoever has to maintain your "one-time" solution will thank you, and you can never have too much good karma....

There's a lot of room between those two ends of the data processing spectrum and that's where these techniques come into their own. A typical situation is one in which:

- you can break the problem into simple transformations, each of which can be expressed in just a few lines;
- it's easy to check that your output is correct, so you don't need the kind of exhaustive unit testing you really should do in most cases;
- you're building a prototype of a more sophisticated solution (e.g., one element of a larger data processing pipeline that you'll replace as soon as Bob gets back from vacation and finishes what he was working on);
- enterprise-scale infrastructure support isn't an issue (i.e., you don't have to support your solution on hundreds or thousands of servers for several years); or
- your problem is I/O bound (i.e., the speed of your disk, your database, or your network dominates processing time), so a simple program will in practice run just as fast as a clever one.

1.5 Road Map

There's a lot of data out there crying out for crunching and a lot of different ways to crunch it. We'll cover the most common formats and techniques in the following order:

- Chapter 2, *Text*, on page 8: how to handle plain ol' text files and (more important) the general principles that underpin every well-behaved data crunching program.
- Chapter 3, *Regular Expressions*, on page 39: how to work with regular expressions, which are the power tools of text processing.
- Chapter 4, *XML*, on page 74: how to handle HTML and XML. We'll look at the three most common approaches: processing data as if it were a stream of tags, processing it as if it were a tree, and using a specialized language called XSLT.
- Chapter 5, *Binary Data*, on page 114: how to process binary data, such as images and ZIP files.
- Chapter 6, *Relational Databases*, on page 134: this is where data crunching blends into full-blown enterprise-scale data processing. I'll show you the

10% of SQL that will account for 90% of the queries you'll ever need to write, and warn you about a few common traps.

- Chapter 7, *Horseshoe Nails*, on page 164: a few topics that didn't fit elsewhere. The most important one is how to test data crunching programs, but I'll also cover some common data encoding schemes and how to handle dates and times.

Topics I *won't* cover are those specific to scientific number-crunching, particularly statistics and data visualization. As a one-time computational scientist, I think these are important, but they are complex enough to deserve a book of their own.

Acknowledgments

My name might be the only one on the spine of this book, but I had a lot of help writing it. In particular, I want to thank

- Brian Kernighan and his coauthors, for the books that taught my generation how to program ([KR98], [KP84], [KP81], and [KP78]);
- Irving Reid, Harald Koch, Gene Amdur, and the rest of the Select Access team, for teaching me the "other 90%" of software development;
- Prof. Diane Horton, and everyone else at the University of Toronto, for letting me put these ideas in front of impressionable young minds;
- Andy Hunt and Dave Thomas, for letting me talk them into this;
- the production team—Kim Wimpsett and Jim Moore—for turning electrons into pages;
- my wonderful reviewers (David Ascher, Michelle Craig, Elena Garderman, Steffen Gemkow, John Gilhuly, Brent Gorda, Adam Goucher, Mike Gunderloy, Alex Martelli, Jason Montojo, John Salama, Mike Stok, and Miles Thibault), without whom this book would be ful of misstakes;
- Sadie, for so much more than just homemade cornbread; and
- as always, my father, for teaching me how to write, and that writing well is important.