Practical Perl Tools

CSV and the Spreadsheet Go A-Wanderin'

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

I can't predict this for sure, but if I were a betting man I'd lay even money that at some point in your career you are going to be handed data in CSV or Microsoft Excel format and be asked to parse it. You may even be asked to produce data in one of those formats.

CSV, which can either mean "comma-separated" values or "character-separated" values, depending on whether you woke up on the pedantic side of the bed in the morning, is one of the more ubiquitous data formats on the planet. Similarly, there are people in the business world who treat Excel as their mother tongue, so being able to read and write spreadsheets in this format will definitely make you a hit among the suits. By the way, I am aware there are more open and liberty-leaning alternatives to Excel around. Much of what we'll talk about in this column will still be useful if those alternatives are your tools of choice, but I mostly won't be addressing them directly.

C D CSV

The canonical, most basic module for processing CSV in Perl is Text::CSV. If you are going to use this module and there are no pure-Perl restrictions in place, you will definitely want to also install the Text::CSV_XS module at the same time. Text::CSV_XS is a replacement backend for Text::CSV written in C that is much, much faster than the pure-Perl parsing routines found in Text::CSV. These two modules are perfectly intertwined: Text::CSV will automatically use Text::CSV_XS with no change of syntax if it notices it is available.

You might wonder why someone has gone to the trouble of writing an entire module to parse CSV data when it seems as though a few simple split() functions would do the trick. Take a moment to read the documentation for the module. Right around the time your eyes start to glaze over from reading all of the possible options, you'll probably come to the conclusion, "Hey, the CSV format isn't as simple as I thought. There are many different ways it could be implemented." And, indeed, I've talked to Perl programmers who have related horror stories about how different program/systems they had to mesh had slightly different interpretations of how CSV should be parsed/written. Using Text::CSV and its copious options can help to shield you from this unpleasantness.

Let's look at the basics of how Text::CSV is used, check out one advanced feature, and then look at a few other more sophisticated modules that use Text::CSV to do the dirty work. To use Text::CSV, we start with code that looks like this (slightly modified from the example in the doc):

```
use Text::CSV;
use Data::Dumper;

my $c = Text::CSV->new( { binary => 1 } )
       or die Text::CSV->error_diag();

open my $FILE, '<', $ARGV[0] or die "Can't open $ARGV[0]: $!\n";

# row will be an array ref pointing to an array of parsed fields
while ( my $row = $c->getline($FILE) ) {
       print Dumper($row);
}
$c->eof or $c->error_diag();
close $FILE;
```

Here's one section of output showing my local airport when I run it against the airports.csv file provided by OurAirports at http://www.ourairports.com/data/:

```
$VAR1 = [
          '3422',
          'KBOS',
          'large_airport',
          'General Edward Lawrence Logan International Airport',
          '42.36429977',
          '-71.00520325',
          '20',
          'NA',
          'US',
          'US-MA',
          'Boston',
          'yes',
          'KBOS',
          'BOS',
          'BOS',
          'http://www.massport.com/logan/',
          'http://en.wikipedia.org/wiki/Logan_International_Airport',
          'General Edward Lawrence Logan International Airport'
        ];
```

Let's walk through the Text::CSV code above so you can see how this all works. After loading the module (with Text::CSV_XS being loaded implicitly), we create a parser object using new(). There are quite a few options available but in this case I'm setting "binary," the one option that I think makes sense to include in every Text::CSV script you write (I'd be hard-pressed to think of a case where you wouldn't want it). Using the binary option will allow the parser to parse fields that contains non-ASCII characters without choking. You never know when José Feliciano or Björk Guðmundsdóttir might show up in your data, so it is safer to set that option.

The code then opens up a file handle using the preferred three-argument form of open(). From this file handle, we have Text::CSV read and parse each line using getline(). The getline() function reads a line, then parses it into fields, with each field occupying a position in an array. It returns a reference to this array, the contents of which we dump out in the loop. The getline() call will return "undef" if it

can't read any more data or if the parsing process crashes and burns. This explains the line after the loop that says:

```
$c->eof or $c->error_diag();
```

If getline() exited because it hit the end of file, everything is peachy (and the first part of that statement is true so it shortcuts to the next line). If not, we call a function that will print the failure information to STDERR. We close the file and go home, a job well done.

I dumped the contents of the array within the loop, but you can imagine doing all sorts of intricate and productive things by accessing $row->[$someposition]. If we want to get a little more sophisticated with Text:CSV we can use getline_hr() instead of getline() to return a hash reference along the lines of:

```
$VAR1 = {
          'iso_country' => 'US',
          'municipality' => 'Boston',
          'home_link' => 'http://www.massport.com/logan/',
          'local_code' => 'BOS',
          'keywords' => 'General Edward Lawrence Logan International
Airport',
          'iata_code' => 'BOS',
          'latitude_deg' => '42.36429977',
          'iso_region' => 'US-MA',
          'id' => '3422',
          'longitude_deg' => '-71.00520325',
          'name' => 'General Edward Lawrence Logan International Airport',
          'elevation_ft' => '20',
          'ident' => 'KBOS',
          'wikipedia_link' => 'http://en.wikipedia.org/wiki/Logan_
              International_Airport',
          'scheduled_service' => 'yes',
          'continent' => 'NA',
          'type' => 'large_airport',
          'gps_code' => 'KBOS'
        };
```

It is soooo much easier to read code that says:

```
$row->{'longitude_deg'};
```

instead of:

```
$row->[5];
```

In order for getline_hr() to work its magic, it has to know what each of the fields represents. To set this information, we have to call column_names() with an array reference to an array with the list of field names prior to calling getline_hr(). In our case, we were working with a .csv file that had a descriptive header row at the beginning of the file:

```
"id","ident","type","name","latitude_deg","longitude_deg","elevation_ft",
"continent","iso_country","iso_region","municipality","scheduled_service",
"gps_code","iata_code","local_code","home_link","wikipedia_link","keywords"
```

so we could include this right before our parsing loop changed to use getline_hr() instead of getline():

```
my $header = $c->getline($FILE);
$c->column_names($header);
```

or, if we wanted to be terser, just:

```
$c->column_names( $c->getline($FILE) );
```

If you decide neither an array ref nor a hash ref floats your boat, you can take a page from the DBI playbook (truth be told, I don't know which came first) and use bind_columns() instead. With bind_columns() your code looks like this:

```
$c->bind_columns (\$id, \$ident, \$type, ... \$keywords);
while ($csv->getline ($FILE)) { ... }
```

Each time through the loop, the data is parsed and then assigned to each of those scalars in order.

Earlier in this section I suggested that there might be more sophisticated modules for each task. In the case of Text::CSV there are quite a few modules that build upon it. Let's look at two that try to make it even simpler to use for the most common cases:

1. Text::CSV::Simple attempts to collapse down the code we saw before into a very simple set of lines:

```
use Text::CSV::Simple;
my $c = Text::CSV::Simple->new( { binary => 1 } );
my @rows = $c->read_file( $ARGV[0] );
```

At this point `@rows` is an array of arrays. Well, more precisely it is an array containing references to other arrays that represent the rows. The row arrays have one field per position. But all of this is easier if you think of it as an array of arrays, so that `$rows[0][2]` would refer to the third field in the first header row (the string "type").

If we didn't want to capture all of the fields in each row, we could instead use the want_fields() method to specify just the field numbers desired. Text::CSV::Simple can also do the equivalent trick with a hash reference if you use the field_map() method:

```
$c->field_map(
    "id",             "ident",
    "type",            "name",
    "latitude_deg",   "longitude_deg",
    "elevation_ft",   "continent",
    "iso_country",    "iso_region",
    "municipality",   "scheduled_service",
    "gps_code",       "iata_code",
    "local_code",     "home_link",
    "wikipedia_link", "keywords"
    );

my @rows = $c->read_file( $ARGV[0] );
```

Now `@row` contains hash references instead of array references, so you can say:

```
# we could leave out the arrow, but it is harder to read
$row[1]->{ident};
```

To perform the equivalent function of want_fields() in the last example, we could specify "null" in the field_map() statement, as in:

```
$c->field_map('id', null, null, 'name', ... );
```

and those fields will simply be ignored in each row.

2. If you liked Text::CSV::Simple's ability to grab data and place it into a data structure in one fell swoop, but didn't like that it changed data structures based on whether another method had been run before it, you might like Text::xSV::Slurp better. It gets used like this (to mimic functionality we've seen so far):

```
use Text::xSV::Slurp 'xsv_slurp';

open my $FILE, '<', $ARGV[0] or die "Can't open $ARGV[0]:$!\n";

my $aoa = xsv_slurp( $FILE, shape => 'aoa',
                            text_csv => { binary => 1 } );
```

The key magic here is the "shape" parameter. That can be one of these:

```
aoa - array of arrays
aoh - array of hashes
hoa - hash of arrays
hoh - hash of hashes
```

We've seen the first two before; the third allows you to essentially invert the data so that there is a hash that uses the name of the column (from the header information) as a key and a list of all of the values in that column for the value of that hash element. The "hoh" shape lets you pick arbitrary columns to use as keys in an (often multiply nested hash) data structure. It also allows you to provide code to handle "non-unique key combinations." For all of these shapes, the module gives you the opportunity to specify code for col_grep and row_grep parameters that will be used to select certain columns or rows for inclusion in the data structure returned by xsv_slurp().

Are you getting tired of CSV modules yet? Me too, so let me wrap up a few small details and we can move on. The first is that all of the modules we've been discussing have "un-parse" methods that will take a data structure of some sort (e.g., an array) and collapse it into a CSV row for writing. There's nothing very sophisticated in how they work, so I'll pass on providing an example. The second detail I'd be remiss if I didn't mention is that there are other alternatives to Text::CSV-based modules that are worth exploring. For example, Text::xSV makes it easier to cope with CSV data that contains newlines in a field (technically allowed, but usually a pain if you have to deal with it). And finally, there are several modules designed to just slurp CSV files right into databases (e.g., DBIx::TableLoader::CSV) that you should consider if that happens to be your use case.

And Now, the Spreadsheet: Reading

Let's visit our second data format. Despite any misgivings you might have about Microsoft and its business practices, and no matter how much you've tried to avoid sullying your hands by touching only non-proprietary formats, your delicate ocular nerves must have at one time or another lost their innocence and gazed upon an Excel spreadsheet. Or perhaps you think Excel is the coolest thing since Daedalus's

wings and you absolutely adore Excel. No matter where on the continuum you find yourself, at some point you may still be called upon to either read or write an Excel spreadsheet. We're now going to explore how this can be done from Perl.

One quick plot twist before we get there: Excel spreadsheets come in two basic flavors. Prior to Excel 2007, the file format was a relatively intricate binary format (Wikipedia tells me this was called "BIFF," for Binary Interchange File Format, so hey, learn a new thing every day). This was the format of the beloved .xls extension. After that version, Microsoft switched to an XML format (albeit compressed essentially into a zip file) that used an extension of .xlsx by default instead. Although it might be a bit informal or imprecise to refer to the file format by its extension (i.e., .xls-formatted), I'm going to do so because I think it is clearer.

The old .xls format is still opened seamlessly by the newer versions and most anything that processes Excel spreadsheets. The Perl module landscape has far more modules to deal with .xls files than it does with .xlsx files. You can assume that the modules we'll be talking about deal primarily with .xls files unless I mention otherwise.

Just as Text::CSV was a core CSV-parsing module, I think it is safe to say that Spreadsheet::ParseExcel can be considered the equivalent for Excel spreadsheets. The example from the documentation does an excellent job of demonstrating the basic operating principles of the module:

```
use Spreadsheet::ParseExcel;

my $parser = Spreadsheet::ParseExcel->new();
my $workbook = $parser->parse('Book1.xls');

if ( !defined $workbook ) {
    die $parser->error(), ".\n";
}

for my $worksheet ( $workbook->worksheets() ) {

    my ( $row_min, $row_max ) = $worksheet->row_range();
    my ( $col_min, $col_max ) = $worksheet->col_range();

    for my $row ( $row_min .. $row_max ) {
        for my $col ( $col_min .. $col_max ) {

            my $cell = $worksheet->get_cell( $row, $col );
            next unless $cell;

            print "Row, Col = ($row, $col)\n";
            print "Value = ", $cell->value(), "\n";
            print "Unformatted = ", $cell->unformatted(), "\n";
            print "\n";
        }
    }
}
```

Basically, we create a new parse object and point it at an .xls-formatted file. In return, we get a workbook object (if not, we bail). From that workbook we can further home in on a specific worksheet. Using the worksheet object, its methods allow us to determine the row and column ranges present in that worksheet. We then iterate over the rows and columns, retrieving a specific cell as we move

through it. For each cell object, we can retrieve its value and also examine how that cell is formatted. This workbook->worksheet->cell sort of approach to dealing with spreadsheets is also present in Spreadsheet::XLSX, the .xlsx equivalent to Spreadsheet::ParseExcel. The documentation says, "It populates the classes from Spreadsheet::ParseExcel for interoperability; including Workbook, Worksheet, and Cell," but its example code shows it can also support a slightly different syntax.

Based on our experience with CSV-parsing modules, you can probably guess that there exist other modules slightly higher up in the food chain designed to make working with these parsers easier. The first of them has the entirely predictable name of Spreadsheet::ParseExcel::Simple. Working with the ::Simple version consists of the following model, according to the documentation: "You simply loop over the sheets, and fetch rows to arrays." Like so (again, from the docs):

```
use Spreadsheet::ParseExcel::Simple;
my $xls = Spreadsheet::ParseExcel::Simple->read('spreadsheet.xls');
foreach my $sheet ( $xls->sheets ) {
    while ( $sheet->has_data ) {
        my @data = $sheet->next_row;
    }
}
```

A second module that attempts to provide a simpler interface deserves mention, not for the module itself but for a sample script that ships with it that has tremendous utility. Spreadsheet::BasicRead comes with xslgrep.pl, a script that will search all contents of all of the cells in the .xls-formatted spreadsheets in a directory hierarchy for a specified regular expression—tremendously helpful if you know you have the information embedded in a spreadsheet someplace but can't recall which file it is in.

Spreadsheet::ParseExcel::Stream is a module that may come in handy if you are parsing very large spreadsheets. By default, Spreadsheet::ParseExcel will suck a spreadsheet into memory as part of its parsing process. There's a whole section in its doc, called "Reducing the memory usage of Spreadsheet::ParseExcel," which describes a slightly more complicated way of using Spreadsheet::ParseExcel that does not retain this behavior and its drawbacks. Spreadsheet::ParseExcel::Stream implements that recommendation and provides a simpler interface to boot.

And, finally, one last Excel-reading helper module to end this subsection: Spreadsheet::Read. Spreadsheet::Read attempts to be one interface to rule them all. If you point it at an .xls-formatted file, it will call Spreadsheet::ParseExcel, .xlsx: Spreadsheet::XLSX, .csv: Text::CSV_XS, and, yes, open source fans, .ods (OpenOffice format) files will be parsed by a module we haven't discussed, Spreadsheet::ReadSXC. See the documentation for various fiddly options that can be set. If you are a big fan of "single interface" modules, this module may please you.

And Now, the Spreadsheet: Writing

In my experience, it is far more common to be asked to parse Excel spreadsheets created in Excel than it is to be asked to actually produce those documents. Still, that need also arises on occasion. For those requests, there is a similar wolf pack of modules. Unlike our previous problem domains, this is one place where there isn't a clear base module that can be considered the one true central module everything else uses. As far as I can tell, there are two: Spreadsheet::Write (and its fork,

Spreadsheet::Wright—more on that in a minute) and Spreadsheet::WriteExcel. Both of these are used for writing .xls-formatted files. The choice for .xlsx files is a little clearer: Excel::Writer::XLSX.

Here's how you can choose between Spreadsheet::Write/Spreadsheet::Wright and Spreadsheet::WriteExcel. If you care about being able to write not only .xls files but also OpenDocument (i.e., OpenOffice/LibreOffice) files, Spreadsheet::Wright will be your best bet. If you find that having example code will be useful to your development process, you will want to choose Spreadsheet::WriteExcel because it ships with 80+ samples (Excel::Writer::XLSX, by the same author, ships with a measly 64 example scripts).

My inclination is to use the latter module, because I appreciate distributions that have that superior level of documentation, especially when I am pressed for time. Also, Spreadsheet::WriteExcel plays nice with Spreadsheet::ParseExcel (same author—John McNamara clearly rocks). In the Spreadsheet::ParseExcel distribution there is a Spreadsheet::ParseExcel::SaveParser module that allows you to "rewrite an existing Excel file by reading it with Spreadsheet::ParseExcel and rewriting it with Spreadsheet::WriteExcel."

Let's see a teeny Spreadsheet::WriteExcel example from its doc so you can get a quick sense of how one goes about creating an Excel spreadsheet from Perl:

```
use Spreadsheet::WriteExcel;

# Create a new Excel workbook
my $workbook = Spreadsheet::WriteExcel->new('perl.xls');

# Add a worksheet
$worksheet = $workbook->add_worksheet();

# Add and define a format
$format = $workbook->add_format();    # Add a format
$format->set_bold();
$format->set_color('red');
$format->set_align('center');

# Write a formatted and unformatted string, row and column notation.
$col = $row = 0;
$worksheet->write( $row, $col, 'Hi Excel!', $format );
$worksheet->write( 1, $col, 'Hi Excel!' );

# Write a number and a formula using A1 notation
$worksheet->write( 'A3', 1.2345 );
$worksheet->write( 'A4', '=SIN(PI()/4)' );
```

It is basically taking the process we saw in parsing an Excel file and throwing it into reverse. Create a workbook, add a worksheet to it, and then create cells with certain values and formats.

Although we are basically out of time, I will mention that there are a few modules, such as Spreadsheet::SimpleExcel, that attempt to make creating simple Excel spreadsheets easier. There are also single-task modules such as Spreadsheet::WriteExcel::FromDB for converting database tables into spreadsheets. I encourage you to try out any of these Excel creation modules, because being able to create spreadsheets on the fly from Perl is a neat trick that is sure to impress your coworkers. Take care, and I'll see you next time.