DAVID N. BLANK-EDELMAN

practical Perl tools: making stuff up

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 be 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

BEING THE TRUTHFUL SORT OF SYSadmin (you know, the kind that would have the System Administration Code of Ethics tattooed on my thigh if the process wasn't a bit on the painful side), I don't usually find myself intentionally fabricating false data. But sometimes that's entirely appropriate. We'll look at a case where this is true and how Perl can help with an endeavor that isn't nearly as nefarious as it sounds.

The scenario I have in mind is where you test something you built or configured with "real" data before you let it get anywhere near the production data. For example, you might want to test a database, a data processing script, a Web application, etc., with pseudo-real data. In those cases, it is easy to hand-create ten or even 100 sample records, but that won't tell you much about how your code or server will handle your *N*-million-record production data. One way to get a comparable data set is to create it yourself, and that's where Perl comes in.

Data::Generate

There are (at least) three Perl modules that can help with this task. We're going to look at all three because they all do things in slightly different manners. This should give you a few options, making it more likely you'll find a good one when you next find yourself in this situation. The first module I'd like to show you is potentially the most flexible out of the box but may, paradoxically, not be the most useful one.

Data::Generate works like this: you give it the type or types of data you want generated (i.e., a string, integer or float number, date, time, etc.), a specification for that data, and the percentage of that type (if more than one type is being requested), and it will attempt to hand you back a data set that satisfies this request. One of the neatest things about this and the other module (Data::Maker) that supports similar functionality is the format of the specification. Data::Generate expects you to provide the specification in something that looks like a subset of Perl's regular expression syntax. We're used to using regular expressions as a selection mechanism; in this case imagine you could run things in reverse and have them produce output instead of filter it. For example, if you had the regular expression:

```
# match all 5-characters strings containing a to z, A to Z and 1, 2, and 3
/[a-zA-Z123]{5}/
```

the equivalent Data::Generate code would be:

```
use Data::Generate;
my $dg_rule      = q { STRING [a-zA-Z123]{5} };
my $dg_generator = Data::Generate::parse( $dg_rule ) or die $!;
# get a 10 element data set
my $Results      = $dg_generator->get_unique_data(10);
```

That second line of code requests a data set consisting of five character strings using characters from the specified character class. When we run this code, the variable $Results contains a reference to an anonymous array of the specified length (10) with contents like this:

```
x $Results
0 ARRAY(0xb1c8d4)
   0  'UK1k3'
   1  'UK1l3'
   2  'UKNk3'
   3  'Uj1k3'
   4  'UjNk3'
   5  'lK1k3'
   6  'lK1l3'
   7  'lKNk3'
   8  'lj1k3'
   9  'ljNk3'
```

If you are curious how many unique possible values could be generated given your specification, Data::Generate can tell you:

```
print $dg_generator->get_degrees_of_freedom(); # prints '503284375'
```

Earlier on I mentioned that you could ask Data::Generate to produce a data set with multiple types. Here's some example code of that from the documentation:

```
use Data::Generate;

# generate varchar data with 2 kinds of values:
#   -> 36% values like ( 12222,15222, ...)
#   -> 64% values like (AAXQ,BAXQ,...)
my $input_rule = q {
      VC(24)   [14][2579]{4}      (36%)
               | [A-G]{2}[X-Z][QN] (64%)
      };

my $generator  = Data::Generate::parse($input_rule);
my $Data       = $generator->get_unique_data(10);
```

Data::Generate is also happy to take a date specification that will yield ranges of dates, like (again from the doc):

```
my $input_rule = q {
      DATE '1999' 'nov' [07,thu-fri] '09' : '09' : '09'
};
my $generator  = Data::Generate::parse($input_rule);

#   ->   returns a set of date values (format 'YYYYMMDD HH:MI:SS')
#        corresponding to the 7th and all Thursdays and Fridays
```

```
#       of November 1999.
my $Data= $generator->get_unique_data(10);
```

One last cool feature of this module: in addition to standard types such as STRING, INTEGER, VARCHAR, this module also allows you to provide a "filehandle" type. It will then attempt to read the file listed in that type and suck in all of the values in that file. The cool part is that it will only suck in the data values from the file that match the regexp-like specification you provided.

So why isn't this module the bee's knees? Well, it might be for you if you are happy with the random nature of the data it will be returning. If you don't mind operating with data that doesn't look even remotely real, then you may want to brush your palms off a couple of times and just call it a day. But if you are creating a Web application that will display street addresses or people's names, then this module will provide you with Web screens that list people and addresses from Mister Mxyzptlk's dimension.

Data::Maker and Data::Faker

Here's where Data::Maker and Data::Faker come in handy. Both were created (apparently independently of each other) to generate data that is more "realistic" than the random stuff you would get out of Data::Generate unless you were pre-seeding its data. Data::Maker and Data::Faker are pretty similar in their approach. If asked to pick one of the two to use, I'd probably suggest using Data::Maker because it is actively maintained (Data::Faker was last updated in 2005) and more feature-full. I mention both because Data::Faker has a slightly different set of data it is prepared to mock up out of the box. If for some reason you needed that kind of data, and you needed it in a hurry (vs. writing custom code Data::Maker could use), you could consider using Data::Faker instead. Given this preference, let's take only a quick glance at Data::Faker before we explore Data::Maker in depth. To use Data::Faker, you write code like this (example from its doc):

```
use Data::Faker;

my $faker = Data::Faker->new();

print "Name:      ".$faker->name."\n";
print "Company:   ".$faker->company."\n";
print "Address:   ".$faker->street_address."\n";
print "           ".$faker->city.", ".$faker->state." ".$faker->zip."\n";
```

Data::Faker also has a number of plug-ins, such as Data::Faker::Internet, that expand the kind of data it can produce. For example, if you load it like this instead:

```
use Data::Faker qw(Internet);
```

you would be able to write:

```
# return a fake domain name
print "Domain Name:" . $faker->domain_name . "\n";
# return a fake IP address
print "IP Address: " . $faker->ip_address  . "\n";
```

Let's move on to Data::Maker so we can see how this general idea can be improved. Like Data::Faker, it has a number of plug-in–like modules that ship with the base package. We'll talk about those in a moment.

The Data::Maker interface is a bit different and slightly more complex than the previous two we've seen. Like many other modules, you give it a full

description of just what you want it to do as part of the object constructor (Data::Maker->new) and then use an iterator to request the generated data, one record at a time. Let's start with the specification.

The first thing we specify is the different fields we want our generated record to contain. Like our first Data::Generate example, we can specify this field using a regexp-like description:

```
{name => 'zipcode', format =>'\d\d\d\d\d'},
```

This says there will be a field called "zipcode" whose format consists of five digits. The only other argument we'll want to give is the number of records we hope to generate:

```
record_count => 10000,
```

Once we have a Data::Maker object, we can request data from it:

```
while ( my $record = $dm->next_record ){
  print $record->zipcode->value . "\n";
}
```

In context, the code looks like this:

```
use Data::Maker;

my $dm = Data::Maker->new(
  fields => [
    {name => 'zipcode', format =>'\d\d\d\d\d'}
  ],
  record_count => 10000,
);

while ( my $record = $dm->next_record ){
  print $record->zipcode->value . "\n";
}
```

The use of an iterator here instead of just a function that returns all of the data (like Data::Generate uses) makes a great deal of sense when you start to generate huge data sets. Data::Generate's get_unique_data() is guaranteed to eat memory like it is going out of style if the data set gets big (this is one of several memory-chewing implementation issues it has, perhaps a good reason to avoid it for some people). Data::Maker's next_record() will not have this problem.

You might notice that the code called $record->zipcode->value is used to get the generated zipcode instead of just $record->zipcode. That's necessary because we get back a Field object through which we'll access the value of the specific field. The author of the module tells me he'd like to make the $record->zipcode code return just the value in a scalar context (because that's what people want most of the time), but that's not yet in the module.

If this were all Data::Maker did, it wouldn't be all that interesting. More interesting is its plug-in–like system (although it doesn't call them plug-ins). With this system, you can write modules or use the author's provided submodules to extend the number of data types that are available. Each module provides additional classes that will produce data. Here's a sample of this from the documentation:

```
use Data::Maker;
use Data::Maker::Field::Person::LastName;
use Data::Maker::Field::Person::FirstName;
```

```
my $maker = Data::Maker->new(
  record_count => 10_000,
  delimiter => "\t",
  fields => [
    {
      name => 'lastname',
      class => 'Data::Maker::Field::Person::LastName'
    },
    {
      name => 'firstname',
      class => 'Data::Maker::Field::Person::FirstName'
    },
    {
      name => 'phone',
      class => 'Data::Maker::Field::Format',
      args => {
        format => '(\d\d\d)\d\d\d-\d\d\d\d',
      }
    },
  ]
);

while (my $record = $maker->next_record) {
  print $record->delimited . "\n";
}
```

In this sample, you can see three fields being defined using classes, each being provided by either Data::Maker or a secondary module that is loaded. The third field defined (phone) is very similar to zipcode in our last example except that it is being described in a more explicit way by referencing the class and passing in a specific argument. We're also throwing in another nicety in the data set definition by indicating that we'll want records to be returned (by $record->delimited) with fields separated by a tab character.

Data::Maker (as of this writing) ships with additional field types that include dates/times, first/last names, middle initials, social security numbers, "random" text (Lorem Ipsum, courtesy of Cicero), and file contents (so you can pre-seed the data). To make this even cooler, the author has also provided data types that can depend on other fields. For example, if you use code like this in your field definition:

```
{
  name => 'initials',
  class => 'Data::Maker::Field::Initials',
  args => {
  from_field_set => [ 'firstname', 'lastname']
}
```

the initials field will be filled in based on the first name and last name fields. If you use the Data::Maker::Field::Person::Gender class, it will attempt to guess in a mechanical fashion the gender of the fake person based on their first name. Anyone who has any experience with gender issues knows that humans can't get their act together around gender, so I wouldn't expect very much from that guess. Still, it is cool that the module makes provision for creating internally consistent fake data based on relationships between fields. It is pretty clear after talking to the module author that he is dedi-

cated to improving the module and adding more excellent features to it in the future.

Now that you have a cool tool for creating a bunch of fake data, let's draw this column to a close. Take care, and I'll see you next time.