DAVID N. BLANK-EDELMAN

practical Perl tools: random acts of kindness

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

IF YOU SUBSCRIBE TO THE NOTION THAT it is a good thing to practice random acts of kindness, how exactly do you ensure they are random? In this column, we're going to take a pseudo-random walk through a number of ways people interact with randomness using Perl and various Perl modules. You'll encounter randomness in many application domains, including cryptography, data protection, software testing, Web development, voting, games, statistics, to name just a few. Rather than focusing on any one domain for use, this column will point at some tools to make your use of randomness easier.

Perl's Built-in Functions

The best place to start talking about randomness in Perl is with its two random number–related internal functions: rand() and srand(). The former provides you with access to your operating system's random number generator (which is either good or bad, depending on your operating system). By default, it returns "a random fractional number greater than or equal to 0 and less than the value of an optional argument." You'll often see code that looks like this:

```
print int(rand(20));        # prints a random integer
                            from 0 to 19
```

Some random number generation algorithms produce "more random" numbers than others. Which algorithm your operating system uses isn't always obvious or documented, hence my comment above. The one thing you can say about all of them is that in order to get the best possible (or even correct) use of rand(), you have to first wisely choose an initial value to "seed" it with. This is where the srand() function comes into play.

The srand() function accepts a seed value that rand() will use. If you give srand() the same value each time, rand() will generate the same "random" numbers each time. That repeatable quality in a random number generator may seem a bit strange, but we'll discover at least one use for it later in this column.

rand() calls srand() by default the first time it is called. srand() tries to use a decent default seed value "based on time of day, process ID, and memory allocation, or the /dev/urandom device if available," but you still see professionally paranoid

programmers calling this function when they want to be extra careful. We'll see one module in the very next section that some people use to super-duper-secure seed their random number generator.

Better Generators

If you are not satisfied with using rand() to access the OS's built-in random number generator, Perl offers a ton of alternatives. I should note that I'm not a mathematician (nor do I play one on TV), so I am not qualified to recommend one random-number-generator algorithm/module over another. I can point to a few that seem popular and talk a good game, but if you are going to need to use one because you have a serious need for the most "secure" RNG (random number generator) available, best to talk to someone more serious about this stuff than I am. Three modules in this vein are:

1. BSD::arc4random—to make use of the same RNG algorithm used by OpenBSD and others
2. Math::Random::MT (and related modules)—to use the Mersenne Twister RNG (though MT RNGs produce "high-quality" random numbers they are known not to be suitable for crypto uses)
3. Math::Random::ISAAC—to use the ISAAC RNG (still apparently crypto-safe)

Let's look at how to use the last two just so you get a feel for how they work.

The easiest way to use a Mersenne Twister RNG is through the Math::Random::MT::Auto module (there is a Math::Random::MT module, but this one has the added feature of making it easier to seed the RNG from a number of sources). To use Math::Random::MT::Auto, your code can be as simple as:

```
use Math::Random::MT::Auto 'rand';
# Perl's built-in overridden with an MT
```

This one line effectively substitutes the built-in rand() function with one backed by the MT algorithm. If that sort of overloading magic gives you the heebie-jeebies, you can be more explicit in how it gets used:

```
use Math::Random::MT::Auto;
my $rng = Math::Random::MT::Auto->new();
print $rng->irand(20)     # mimics the rand() example above
                          # use ->rand() instead for a rand() clone
```

Math::Random::MT::Auto also offers bonus functions like shuffle() to shuffle arrays in an MT-inspired fashion.

Math::Random::ISAAC (and its accompanying fast version, Math::Random::ISAAC::XS) is similarly easy to use:

```
use Math::Random::ISAAC;
my $rng = Math::Random::ISAAC::XS->new( @seeds );
$rng->irand();
```

Math::Random::ISAAC makes a point of not trying to pick a good seed value by default because the author believes that's a decision the user should make in a careful and concerted fashion. The documentation does point out a number of ways to generate a good seed value, one of which I want to mention because it addresses the issue we left open before when discussing srand(). At the moment, one of the better ways is to use the Math::TrulyRandom module that attempts to generate values based on "interrupt timing discrepancies." Math::TrulyRandom gets used like this:

```
use Math::TrulyRandom;
$random = truly_random_value();
```

According to the documentation, "The random numbers take a long time (in computer terms) to generate, so are only really useful for seeding pseudo random sequence generators."

Before we move on to the next section, I want to mention one of the more intriguing places you could turn to for randomness, namely the Web. The Perl module Net::Random queries one of two Web-based random number sources. One is fourmilab.ch, home of the HotBits project [1]. According to their site:

> HotBits is an Internet resource that brings *genuine* random numbers, generated by a process fundamentally governed by the inherent uncertainty in the quantum mechanical laws of nature, directly to your computer in a variety of forms. HotBits are generated by timing successive pairs of radioactive decays detected by a Geiger-Müller tube interfaced to a computer. You order up your serving of HotBits by filling out a request form specifying how many random bytes you want and in which format you'd like them delivered. Your request is relayed to the HotBits server, which flashes the random bytes back to you over the Web. Since the HotBits generation hardware produces data at a modest rate (about 100 bytes per second), requests are filled from an "inventory" of pre-built HotBits. Once the random bytes are delivered to you, they are immediately discarded—the same data will never be sent to any other user and no records are kept of the data at this or any other site.

The other random number source is random.org, a site devoted to randomness. Their site says:

> RANDOM.ORG offers *true* random numbers to anyone on the Internet. The randomness comes from atmospheric noise, which for many purposes is better than the pseudo-random number algorithms typically used in computer programs.

Random.org has a quota system in place that makes sure people don't abuse the system and try to consume all of the random bits it produces. You can, however, buy a larger quota, i.e., more bits. It amuses me that it is now possible to figure out how much randomness costs (at the time of this writing: $150 USD will get you 600,000,000 bits, or 4 million bits per dollar, which seems like quite a bargain, no?).

The Net::Random module knows how to query both Web services. Here's an example from its documentation:

```
use Net::Random;
my $rand = Net::Random->new();# use fourmilab.ch's randomness source,
        src => 'fourmilab.ch',          # and return results from 1 to 2000
        min => 1,
        max => 2000
);
@numbers = $rand->get(5);         # get 5 numbers
```

As the documentation for Net::Random points out, there are certainly security concerns with using a service like this (especially when you add caching Web proxies to the mix), so best check out the documentation before you start to use this for a serious project.

Modules for Generating Random Data

That last bit provides a nice segue into a section on ways Perl can help make using randomness in your applications easier. We saw a bit of this in my last column. In that column we explored a number of Perl modules like

Data::Generate and Data::Maker for creating random but plausible-looking data records. Let's look at a further expansion of the theme.

The first set of modules, similar to those we saw last time, are those that take some sort of specification and return random data in the form of your choice. For example, String::Random lets you write code like this:

```
use String::Random;
my $srobj = new String::Random;
$rand_string = $srobj->randregex('\d[a-z]\d');
```

Once run, $rand_string will consist of a random string containing a digit, a letter from a to z, followed by another digit. String::Random can either take a regular expression (using a subset of Perl regexp syntax), as was done above, or take a pattern more like pack() and create random strings based on that specification.

To create a more targeted set of data, you might find Data::Random and Data::Rand::Obscure more useful. The former lets you request different kinds of data using functions like:

```
rand_words()      - produce random words from a list
rand_chars()      - produce random characters from a defined set
rand_set()        - produce random array elements from a given array
rand_date()       - generate random date strings
rand_time()       - generate random time strings
rand_datetime()   - generate random date/time strings
```

and my favorite:

```
rand_image()      - generate a random PNG-format image
```

All of these functions behave the way you would expect—you hand them a few initialization parameters (such as the size of the character string you want to get back and the number of strings to return) and they return the data requested. The one thing you may find Data::Random doesn't do is provide a facility for only returning values not previously returned (i.e., only unique responses). For that you may wish to check out Data::Rand instead. It lets you provide a flag called 'do_not_repeat_index' which keeps the module from using any one array index into the given set more than once.

Data::Rand::Obscure is slightly more focused than either of these modules. Despite its name, it bears a closer resemblance in function to Data::Random than to Data::Rand. Data::Rand::Obscure provides a few functions along the lines of those we saw for Data::Random:

```
create_hex()   - create a random hex string (also create())
create_b64()   - create a random base64 string
create_bin()   - create a random binary value
```

The module "first generates a pseudo-random 'seed' and hashes it using a SHA-1, SHA-256, or MD5-digesting algorithm." The documentation goes on to say, "You can use the output to make obscure 'one-shot' identifiers for cookie data, 'secret' values, etc." It also makes dandy session keys for Web programming.

Make the Randomness Go Away

I tend to get a kick out of modules that are both counterintuitive and solve a clear problem using this twist from their usual expectations. Let's bring this column to a close by looking at two modules that solve basically the same problem.

Here's the issue in a nutshell: sometimes you write Perl code that should be tested using random data as input. Creating such tests is a commendable endeavor, but they add another layer of complexity to the development process. If you find one of your module's tests failing when presented with a certain input (randomly generated or not), you can't really be sure whether later programming efforts have squashed the bug unless you have some way to precisely reproduce the initial input to the code. In this scenario, we need a method for making previously random inputs reproducible (which essentially means removing the randomness from the "random inputs").

Both of these packages can intercept your tests' calls to rand(). In the case of Test::Random, you simply:

```
use Test::Random;
```

in front of your usual test code. If you call your test program with the TEST_RANDOM_SEED environment variable set, your code will use that particular seed every time. By default, Test::Random will display its current random seed so you can feed it back into the program. For example (from the Test::Random documentation):

```
$ perl some_test.t

    1..3
    ok 1
    ok 2
    ok 3
    # TEST_RANDOM_SEED=20891494266

$ TEST_RANDOM_SEED=20891494266 perl some_test.t

    1..3
    ok 1
    ok 2
    ok 3
    # TEST_RANDOM_SEED=20891494266
```

From this example you can see how the Test::Random magic lets you run the same test each time using the same predictable "random" input. Test::MockRandom is slightly more complicated and meant for intercepting rand()-like calls from within object-oriented programs. Be sure to read its documentation for further information on how to actually use it.

Take care, and I'll see you next time.

REFERENCES

[1] http://www.fourmilab.ch/hotbits/.