DAVID N. BLANK-EDELMAN

# practical Perl tools: family man

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 TELL WHETHER I SHOULD apologize more profusely for writing another column about a family of Perl modules or for writing one inspired, at least in name, by a Hall and Oates song. Truth is, I'm partial to both, so I guess you'll just have to hang in there and see which bothers you the least. If it helps any, I'd recommend you check the Chromeo episode at http://www.livefromdarylshouse.com/currentep.html?ep_id=35 for some splendid updates of a few classic hits (including this column's title).

But you are in luck! In this column we're going to be discussing a family of modules that are bound together, not only in name, but also by their exceptional usefulness. We're going to explore a number of modules that modestly call themselves Something::Util(s) when, really, they should be called Something::ModulesYouAreSurprisedYouCouldLiveWithout. Some of the modules we're going to look at aren't shipped with Perl (i.e., "in the core"), but there has certainly been much discussion over the years about including them.

## Working with Lists

Let's start with a core module: List::Util has been in core for eons, but I still encounter people who have never heard of it. I know I came to it relatively late, so perhaps a little reminder of how useful it is will be in order.

List::Util can export the following subroutines for use in your code:

- first—returns the first instance of something in a list
- max/maxstr—returns the element with the highest numerical/string value
- min/minstr—returns the element with the lowest numerical/string value
- reduce—reduces a list (more on this in a second)
- shuffle—returns the elements of a list in random order
- sum—adds up all of the elements in a list

Two of these deserve a little more exploration. The first on the list, first(), comes in handy in a common case that may not be obvious. Many is the time when I've found myself wanting to know if a particular string can be found in any of the elements of a list. For people with a UNIX background, the reptile part of our brain that holds UNIX's semi-obtuse command names jumps in and suggests using grep(), as in:

```
if (grep {/fred/} @list ) { do_something; };
```

The problem with this is grep(), like its command counterpart, wants to search the entirety of its target. If you type "grep fred file" it will search all of file and show you all of the matches. Likewise, no matter how many terabytes of memory long @list is above, grep() will search all of it before returning a result. My task was "are there any?" not "what are all?" matches, so grep() is potentially doing a lot more work than we need. When you point this out to a new programmer, their next shot is a for() loop that contains an if-block with a test and a separate statement designed to exit the loop when it finds the first element that matches. That works, but it is too much code. A more concise version:

```
use List::Util qw(first);

if (first {/fred/} @list ) { do_something; };
```

The second function that deserves greater mention is reduce(). This function (more precisely, the abstract notion of a function with this definition) has been very popular in the trade press and among the big data crowd because it is the second part of the much vaunted MapReduce approach. Here's how the doc describes it:

> Reduces LIST by calling BLOCK, in a scalar context, multiple times, setting $a and $b each time. The first call will be with $a and $b set to the first two elements of the list, subsequent calls will be done by setting $a to the result of the previous call and $b to the next element in the list.

So let's look at a quick example to make this description a little clearer:

```
use List::Util qw(reduce);
print reduce { $a * $b } (1,2,3,4,5); # prints 120
```

How did we get 120?

The reduce() call starts by assigning 1 to $a and 2 to $b. It multiplies them, assigns the result of 2 to $a (1 times 2 is 2) and 3 (the next number in the list) to $b. It multiplies 2 times 3 yielding 6.

This process repeats:

- assigning 6 to $a and 4 to $b, multiplying 6 times 4 to get 24
- assigning 24 to $a and 5 to $b, multiplying 24 times 5 to get 120

The end result is we've reduced the initial list of five elements down to a single result. We're using a very simple reduction operation (multiplication), but that initial block can contain code that manipulates $a and $b in any way you'd like.

List::Util looks like a handy collection of list manipulation functions, but they are all pretty basic. If we are willing to step out of core, we will be welcomed with open arms by List::MoreUtils. List::MoreUtils has quite a few more Utils. Most are of the form:

```
function_name {some piece of code} @list;
```

Let me break them down into some rough categories for you.

- list element membership questions: any, all, none, notall

  This allows you to write things like:

  ```
  use List::MoreUtils qw(any none);
  if (any {/fred/} @list) { do_something };
  if (none {/fred/} @list) { do_something };
  ```

For consistency's sake, I used /fred/ in the code blocks above, but really, that could be any code that makes a decision by returning true or false when presented with each element of the list (via $_). Instead of a basic regexp test, we could use anything:

```
if (any {$_ == 3} @list) { do_something };
if (any {people_want_to_come_out_and_play($_)}) { do_something; };
```

- list member counts: true, false

  These return the number of elements for which the {code} block is true or false, as in:

  ```
  use List::MoreUtils qw(true false);
  my $num_of_fredful_items      = true {/fred/} @list;
  my $num_of_items_lacking_fred = false {/fred/} @list;
  ```

- list member searches: firstidx (first_index), lastidx (last_index), firstval (first_value), lastval (last_value), uniq

  The first set of functions in this category (and their aliases, listed in parens above) will locate the first or the last occurrence in the list where their {code} block becomes true and returns either the index of or the actual value at that spot. The uniq() function will either return a list of the uniq elements in your list or the number of unique elements, depending on how you call it.

- list insertion: insert_after, insert_after_string

  This really nifty function makes for more readable code than the usual push/pop/shift kind of dance used when you want to insert an element into the middle of a list. You tell it a way to locate an element, either by {code} or a simple string, and it will insert the value of your choice into the string right after the first item it finds. For example:

  ```
  use List::MoreUtils qw(insert_after_string);
  my @list = ('sing','a','song');
  insert_after_string 'a','joyous', @list # @list now has sing,a,joyous,song
  ```

- list splitters: before, after, part

  These functions make it easy to split a list into two or more sub-lists. The before() and after() functions will return the parts of a list either before or after the first place its {code} argument is true. The part() function is a bit more sophisticated. It can partition a list into N buckets (sub-lists) based on its {code} argument, returning a list of references pointing to each sub-list. Here's an example adapted from the List::MoreUtils test suite:

  ```
  use List::MoreUtils qw(part);
  my @list = 1 .. 12;
  my $i = 0;
  my @part = part { $i++ % 3 } @list;
  ```

  It yields a list with references to the following sub-lists:

  ```
  [ 1, 4, 7, 10 ]
  [ 2, 5, 8, 11 ]
  [ 3, 6, 9, 12 ]
  ```

  I had to stare at this for a few seconds before I understood what was going on, so let me tell you how this works. It is a little confusing because up until now in this column, {code} always contained a reference to the current element. In this example, we're going to not bother to look at the value of each element and instead will be using a simple counter to calculate where things go.

We create a list whose values are the numbers 1 through 12. We then initialize a counter called $i. The part() function iterates over each element in our list. For each element in the list, we divide the counter value (what is in $i) by 3 and check the remainder. If the remainder is 0, the value of the list at that point will be placed in the first sub-list; if the remainder is 1, it goes in the second sub-list; if the remainder is 2, into the third sub-list, and so on. It goes something like this:

$i = 0, 0 % 3 = 0, so the first element of @list (1) is placed into the first sub-list
$i = 1, 1 % 3 = 1, so the second element of @list (2) is placed into the second sub-list
$i = 2, 2 % 3 = 2, so the third element of @list (3) is placed into the third sub-list
$i = 3, 3 % 3 = 0, so the fourth element of @list (4) is placed into the first sub-list...

- Tweaks: apply, indexes, minmax()

  The first two are small twists on the standard Perl functions of map() and grep(). apply() behaves like map(), except it makes sure you can't perturb the list during the iteration in the same way you can with map(). With map(), if you write:

  ```
  my @results = map {$_ += 5} @list;
  ```

  both @results and @list contain all of the elements of @list with 5 added to them. Some people like to write code that changes a list using some sort of map()'d assignment (not a great idea), and others do it by mistake. If we ran the same code with apply() instead of map(), @list would remain pristine after the operation was over.

  The twist on grep() called indexes() takes the same arguments as grep(), but instead of returning a list of the values found that match {code}, it returns the list of indices where {code} evaluated to true.

  minmax() is more of a tweak on the min() and max() functions we saw in List::Util. It returns a single two-element list with both the minimum and the maximum values found on a given list.

- multi-list functions: mesh (zip), each_array/each_arrayref, pairwise

  This is the final category of functions in List::MoreUtils. These functions let you operate on multiple lists at once. For example, the mesh() function (or its alias, zip()) takes N lists and returns a single list created by collecting the Nth element of each list, followed by the N+1th element, etc. The resulting list consists of the first elements from each of the lists, followed by those elements that were in the second position in each list, followed by those in the third position, and so on. As an aside, if the idea of a zip() operator in Perl sounds vaguely familiar to you, that's because you used your time machine to go into the future where you spied a fairly sophisticated version implemented in Perl 6.

  If you dig how mesh()/zip() work, you'll probably also like each_array() and its variant each_arrayref(). Both of these functions work by creating an iterator for you that will hand back the Nth item from each of those lists. The pairwise() function lets you operate on two lists, a pair of items at a time, using the same {code} block.

  I realize we've been talking about list-related Utils for quite some time, so let me mention one more thing as an outro. It is hard to tell just how tongue-and-cheek this is, but Dave Rolsky says in his List::AllUtils doc: "Are you sick of trying to remember whether a particular helper is defined in List::Util or List::MoreUtils? I sure am. Now you don't have to remember. This module will export all of the functions that either of those two modules defines."

I'm not clear this is a big problem, but hey . . .

## Working with Hashes

By now you are probably sick of list manipulation functions. We'll switch to manipulating a different data type to cleanse your palette. For this section, I want to look at another module found in the Perl core (as of Perl 5.8): Hash::Util. I can't say I use it nearly as often as the previous modules, but it has some functionality that can improve the safety of your code immeasurably under the right circumstances.

Hash::Util has a number of functions that allow you to lock the contents of a hash in different ways. Devoted readers of this column may remember the two-part series on tie()-based modules I did back in August and October of 2006. In that series I mentioned Tie::StrictHash. The functions in this module can operate much like that module, only without all of the funny tie() business. Plus, you get three different granular levels of "locked."

The functions lock_hash() and unlock_hash() operate as you would expect. Once locked, nothing in the hash, either keys or values, can be changed. Any attempt to change anything in the hash produces an error like:

```
Modification of a read-only value attempted at...
```

The next level of locked is lock_keys() and unlock_keys(). If you just call on a hash the way you would on lock_hash(), it will stop the addition of keys to the hash with a message like:

```
Attempt to access disallowed key 'johnnycomelately' in a restricted hash at...
```

Any keys already in the hash when you locked it will be considered "allowed." If you'd like to be more specific about which keys should be allowed besides "anything in the hash at the time of locking," lock_keys can also take a second argument of a list of allowed keys. You can test for presence in a locked hash using exists() as per usual, and can even delete entries using delete(). If you delete a key, you are allowed to put it back into the hash because that key name stays on the "allowed" list.

When you've locked the keys in a hash, you can change the values of any of the allowed keys, which leads us to the final level of granularity: the functions lock_value() and unlock_value() (saw this coming, right?). Code like this:

```
use Hash::Util qw(lock_value);
my %hash = ('dollar' => 1, 'euro' => 1.2, 'yen' => 3);
lock_value(%hash, 'dollar');
lock_value(%hash, 'euro');
lock_value(%hash, 'yen');
```

will make sure that the values for those keys in the hash can't be changed. The ability to lock hashes like this means you can avoid writing code that either puts the wrong keys or the wrong values into a critical array in your program (e.g., with a typo).

Before we switch to the last gear, I should mention that there are a couple of modules worth checking out that can do some nifty things with hashes but that I won't have time to cover here: Deep::Hash::Util offers some easy ways to work with nested hashes, and List::Pairwise (I know, it doesn't end in

::Util, but perhaps it should) lets you do all sorts of neat things with pairs of elements both in lists and in hashes.

## Working with Strings

Old hands at Perl (aye, you scurvy dogs) are probably expecting the final section to discuss utilities associated with scalars, the last remaining major data type in Perl. I would do that except the obvious choice of module for this would be Scalar::Util. The problem is—Scalar::Util is boring, boring, boring. Unless you need to interrogate a scalar to determine if it is blessed, tainted, weak, etc., it won't do very much for you. It's worth knowing that there is a module that can provide this information, but if I had to think of the last time I used Scalar::Util in regular code, I'd probably start reminiscing about the days when computers ran on coal and had to be stoked periodically.

The closest thing to a useful module for this datatype is String::Util. It has a number of syntactic sugar functions like trim (to remove whitespace), crunch (to remove multiple occurrences of whitespace and to trim), nospace (to remove whitespace), and so on. All of those could easily be done with regular expression substitutions, but using a named function may be easier to read.

That's all of the utilities we have time for today. I'd recommend poking around on CPAN for the other modules that have ::Util in their name, because there is some neat stuff there. Take care, and I'll see you next time.