Practical Perl Tools
What's Up, perldoc?

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

For the past million or so columns, we've taken a look at how to use Perl to do X or how to use Perl to do X by communicating with such and such Web service. Every once in a while I think it is good to step back and talk about how to use Perl, period. This is going to be one of those columns where we go back to some of the basics that you may have missed as you zoomed right to the "high priestess of Perl" status you now hold. We're going to talk a bit about documentation in the Perl world, both how to consume and create it.

Rockin' the perldoc

If you looked back at all of the classes I've taught over the years with Perl content in them, I think you could make a safe bet that there would be someone in every class who has never heard of the "perldoc" command. If you really wanted to cash in, you would bet that a large majority of the people in the room who already knew about the command didn't realize everything it can do. Want to take that bet? Read on.

"perldoc" is a command that ships with Perl. It is designed to show you various parts of the Perl documentation installed on your system. The documentation it can display includes all of the text documentation that ships with Perl plus the manual pages for the core modules and any modules you've installed.

So, for example, if you wanted to see the manual page for the File::Spec module, you could type

```
perldoc File::Spec
```

and perldoc would find the documentation, run it through a converter to convert it into man page format, hand the man page to whatever your system uses to display them (e.g., nroff -man), and then show it to you in your favorite pager. This works fine for the larger Perl documentation sections as well:

```
perldoc perldsc
```

(Wait, you mean you didn't know that Perl shipped with such excellent doc as the Perl Data Structures Cookbook and perlperf, the Perl Performance and Optimization Techniques tome? Well, you best type "perldoc perl" right now and then come back to this article in a few hours after you've read some of the good stuff you'll find. For a more verbose version of that listing, try "perldoc perltoc".)

A very reasonable question you might have about this command is "Why not just type 'man File::Spec'?" It's a good question because for the core modules, on many default installations of Perl, this will indeed work. perldoc is preferable for at least two reasons:

1. perldoc will find documentation within a copy of Perl that wasn't installed in the default place ("man" won't find it unless you changed your MANPATH), and
2. unless your Perl was installed carefully with this in mind, non-core modules may install their documentation in a different place or with a different suffix than "man" expects.

But perhaps the best argument for perldoc over man is about to be revealed when we look at the cool stuff it can do.

Not to give away the best hint first, but I don't think I would find coding in Perl as easy as I do if it wasn't for the -f flag to perldoc. The -f flag lets you look up the doc for all of the many, many Perl built-in language functions. Can't remember what the order of the arguments of the split() is? Type:

```
perldoc -f split
```

and you'll see

```
split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
split   Splits the string EXPR into a list of strings and
        returns that list. ...
```

along with all of the rest of the documentation on that function. A similar flag, -v, helps you look up the documentation for a dizzying array of predefined variables in Perl. So let's say you were reading someone else's code and you run into the $(variable. If you didn't want to shake your head sadly and say, "Kids these days, with their wacky emoticons, I just don't understand them..." you could instead type

```
perldoc -v '$('
```

and you'd see

```
$REAL_GROUP_ID
$GID
$(      The real gid of this process. ...
```

For people just starting out with Perl, it can be helpful to type commands such as

```
perldoc -v '%ENV'
```

to see what the %ENV hash is and what it does.

Beginners may be aware that there exists a substantial nine-part FAQ about Perl and how to use it, but I'd dare say that they probably don't know they can search it using perldoc's -q flag. If you type perldoc -q {something}, it will search for that something (using a regular expression search, btw) in the questions text from all of the sections of the perlfaq. If I typed "perldoc -q mail," for example, it would show me the answers to the following questions:

- What mailing lists are there for Perl?
- How do I parse a mail header?
- How do I check a valid mail address?
- How do I return the user's mail address?
- How do I send mail?
- How do I use MIME to make an attachment to a mail message?
- How do I read mail?

I may have listed my most used perldoc hint first, but I have saved the most surprising for last. Very few people know about the -l and the -m flags to perldoc. Here's where they come in handy: anyone who has done any substantial amount of Perl programming has had to go look at the Perl source to a module they are using. You do this for any number of reasons: sometimes it is sheer curiosity for how something has been implemented; sometimes we're struggling to figure out how to use a module and have to resort to the source code for guidance; other times we need to better understand an object it defines and so on.

The first step toward consulting the source code of a module is finding where it lives on disk. This can be done using the -l flag. To return to the very first example, if we wanted not only to see the documentation for File::Spec, but where it was installed, we could type

```
perldoc -l File::Spec
```

and find out this path on OS X's Mountain Lion release:

```
/System/Library/Perl/5.12/darwin-thread-multi-2level/
File/Spec.pm
```

There's a bunch of auxiliary info we're getting back from this little command, including just where modules are installed on the system and the version of Perl in play (or at least the versioned directory presumably associated with that version).

But that's just *where* the file is located; even cooler still is to run perldoc using the -m flag:

```
perldoc -m File::Spec

    package File::Spec;

    use strict;
    use vars qw(@ISA $VERSION);
```

```
$VERSION = '3.31_01';
$VERSION = eval $VERSION;

my %module = (MacOS   => 'Mac',
              MSWin32 => 'Win32',
          os2     => 'OS2', ...
```

Why yes, that is the actual source of the module. With one command we can easily see the source of (the main file of) a module. Very handy sometimes!

perldoc from Orbit

perldoc on your machine works great for providing the documentation for Perl things installed on that machine, but what if you wanted to consult the documentation for a different version of Perl? A lovely resource for that sort of thing is the Web site http://perldoc.perl.org, which has the full doc sets for 16 versions at last count and offers a usable Web interface to boot.

If you fall in love with that Web interface and can't bear to be without it even when you are disconnected from the Intertubes, Jon Allen, the site's creator, offers a module to help you run an HTTP-served version of the doc on any machine. Perldoc::Server will provide a Catalyst-based Web application that can be started up just by typing "perldoc-server". Perldoc::Server will then run a tiny Web server by default on port 7375 ("PERL" on a phone keypad, explains the doc).

So far it appears all of the command line stuff we've talked about displays documentation for things that have been installed locally. That seems kind of limiting. Perhaps you'd like to see the documentation for something you haven't installed on that machine. Pod::Cpandoc will do this for you. When you install it, it provides a command "cpandoc," which can stand in for perldoc if you'd like. If you type

```
cpandoc SomeModule
```

it will display the locally installed doc (just as perldoc would do), and if it can't find it there, it will fetch it right from CPAN. And in case you are curious, the perldoc flags I was crowing about above still work. If you type

```
cpandoc -m SomeModule
```

and SomeModule isn't installed, it will still let you read the source for that module by grabbing it from CPAN. cpandoc even slips in a flag perldoc doesn't have: -c. This flag will show the Changes (i.e., a changelog) for a module if it has one.

Good Documentation Starts at Home

I'd like to switch gears now and move away from how to consume documentation to the question of how to create good documentation for the Perl code you write. The first thing you'll want to do is take a quick look at the Pod (Plain Old Documentation) documentation with a command like "perldoc perlpod". The reason why I say "quick" is I find that reference page to be a bit overwhelming if you've never seen Pod before. Glance over it, maybe make note of the sections on how to embed Pods in Perl Modules and Hints for Writing Pod, but don't get nervous. Pod is described in the doc as "a simple-to-use markup language used for writing documentation for Perl, Perl programs, and Perl modules," and it really is. I think the easiest way to learn Pod is to pick a simple module or command that has been marked up, look at the source, and basically copy what you see there.

For example, if we looked at the source for the cpandoc command line script, we'd see:

```
#!/usr/bin/env perl
use Pod::Cpandoc;
exit( Pod::Cpandoc->run() );

__END__

=head1 NAME

cpandoc

=head1 DESCRIPTION

See L<Pod::Cpandoc> and L<Pod::Perldoc>.

=cut
```

The first part loads the module and calls a function to start it running. But that's not the interesting part for our discussion. After the executable code, there is a marker of __END__ to let Perl's parser know that it has finished finding any code it should read in. From that point on, we see Pod format doc with two headings (=head1), a little bit of body text, and a =cut command to indicate the end of that Pod block. Here we are seeing Pod at the end of the Perl code, but it is also designed to be interleaved with executable code so that the doc is right next to the code it documents. When used with care, this programming style can be used quite effectively. If you find you like the idea of combining code and doc and you'd like to see how far the idea can be taken, I'd encourage you to check out Knuth's work on literate programming (and in case you are curious, I was going use Pod and literate programming in the same sentence until the Wikipedia entry slapped me down hard).

Rather than dwelling on Pod for the entirety of this section, I'd like to end with a look at a spiffy documentation-related module that has actually shipped with Perl since the 5.6 days back in 2000. The Pod::Usage module comes with a pod2usage() function that can do magic if you've embedded Pod documentation

in your code. pod2usage() knows how to find the USAGE and related sections of your Pod documentation and spit them out at a given level of verbosity. You can programmatically decide whether it will show just the USAGE text (i.e., the SYNOPSIS) or even the whole man page. To see all this in action, let's look at the recommended use sample code from the documentation:

```
use Getopt::Long;
use Pod::Usage;

my $man = 0;
my $help = 0;
## Parse options and print usage if there is a syntax
## error, or if usage was explicitly requested.
GetOptions('help|?' => \$help, man => \$man)
       or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-verbose => 2) if $man;

## If no arguments were given, then allow STDIN to be
      ## used only if it's not connected to a terminal
      ## (otherwise print usage)
pod2usage("$0: No files given.")
       if ((@ARGV == 0) && (-t STDIN));
__END__

=head1 NAME

sample - Using GetOpt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

 Options:
   -help            brief help message
   -man             full documentation

=head1 OPTIONS

=over 8

=item B<-help>

Print a brief help message and exits.

=item B<-man>

Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s)
    and do something useful with the contents thereof.

=cut
```

Here we can see a more complete Pod example topped off by calls to pod2usage. If this script gets called with a -man switch, it will show the entire manual page. If it is called with a -help or -? switch, only the SYNOPSIS section will be printed. This is similarly printed if the script doesn't receive the input it expects (i.e., is called with no arguments) as a way of demonstrating how pod2usage() can help provide useful error messages. I think it is a nice touch for a script to be able to supply its own documentation if asked.

So go, document lots. Take care and I'll see you next time.