Perl One-liners

Jeff Bay, jlb0170@yahoo.com

Abstract

This article introduces some of the more common perl options found in command line programs, also known as one-liners. I cover the -e, -n, -p, -M, and -w switches, along with BEGIN and END blocks.

1 Creating a one-liner

Most talented unix Perl programmers I have met have a dirty little secret. They cannot resist the allure of the gnarly Perl one-liner for accomplishing short tasks that do not need a complete script.

The -e switch allows me to write Perl scripts directly on the command line. Code listing 1 shows a simple “Hello world”.

Code Listing 1: Hello World

```
prompt$ perl -e 'print "hello world!\n"'
hello world!
```

In code listing 2, something a bit more complex, I take the output from ls, parse it for the file size, and sum the sizes for all files which are not directories.

Code Listing 2: File size sum

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^dt/; $sum += (split)[4] } print "$sum\n"'
1185
```

I use several tricks in code listing 2. Normally, I do not write something like this all at once. I build it up a bit at a time to make sure I get the correct output at each step. In code listing 3 I check the output of the command.

Code Listing 3: ls output

```
prompt$ ls -lAF
total 32
drwxrwsr-x   2 jbay     staff        512 Feb 21 09:34 adir/
-rw-rw-r--   1 jbay     staff        395 Feb 21 09:29 afile1
-rw-rw-r--   1 jbay     staff        423 Feb 21 09:29 afile2
-rw-rw-r--   1 jbay     staff        120 Feb 21 09:29 afile3
```

In code listing 4 the output from `ls` becomes the standard input to my script, which simply prints each line. I can see that I get the output I expect, the same thing from code listing 3.

Code Listing 4: ls piped to perl

```
prompt$ ls -lAF | perl -e 'while (<>) { print $_ }'
total 32
drwxrwsr-x   2 jbay     staff        512 Feb 21 09:34 adir/
-rw-rw-r--   1 jbay     staff        395 Feb 21 09:29 afile1
-rw-rw-r--   1 jbay     staff        423 Feb 21 09:29 afile2
-rw-rw-r--   1 jbay     staff        120 Feb 21 09:29 afile3
```

In code listing 5 I want to skip the initial total line and directories, so I want to ignore lines that begin with a "d" or "'t". I add a next before the print statements to skip lines that begin with a "d" or a "t", so my program does not print them.

Code Listing 5: Skip lines

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^[dt]/; print $_; }'
-rw-rw-r--   1 jbay     staff        395 Feb 21 09:29 afile1
-rw-rw-r--   1 jbay     staff        423 Feb 21 09:29 afile2
-rw-rw-r--   1 jbay     staff        120 Feb 21 09:29 afile3
```

Once I know that my script skips the right lines, I split the remaining lines and print the value in column 5, the file size. At this point my program, in code listing 6, prints out the same number I see in the ls output, which is the correct size for each file.

Code Listing 6: Print file sizes

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^[dt]/; print +(split)[4], "\n" } '
395
423
120
```

Finally, I want to sum the file sizes. Code listing 7 accumulates the sum in $sum, then prints it at the end of the program.

Code Listing 7: Sum file sizes

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^[dt]/; $sum += (split)[4] } print "$sum\n"'
938
```

Now I have the complex perl one-liner that I showed in code listing 2.

2 One-liner input

Perl programs can receive data from standard input or the command line arguments in @ARGV.

2.1 Standard input

Code Listing 8: Skipping comments

```
prompt$ cat afile | perl -e 'while (<>) { print unless /\s+#/ }'
```

The "|" (pipe) symbol takes the output of cat and makes it the standard input of my Perl program. The diamond operator, <>, reads lines from standard input, so this one-liner reads the lines from afile and then prints the lines that do not match the regular expression `\s+#`.

I can also redirect file contents to perl's standard input using the shell redirection operator, <. Code listing 9 produces the same output as the previous example.

Code Listing 9: Input by redirection

```
prompt$ perl -e 'while (<>) { print unless /\s+#/ }' < afile
```

However, the diamond operator can open and directly read the contents of the file specified on the command line so I do not need to redirect the file contents myself. Code listing 10 does not use file redirection, and does the same thing as code listing 9.

Code Listing 10: Input

```
prompt$ perl -e 'while (<>) { print unless /\s+#/ }' afile
```

2.2 Command line arguments

I can access command line arguments using @ARGV. Code listing 11 simply prints whatever is in @ARGV.

Code Listing 11: Print the command line arguments

```
prompt$ perl -e 'print "@ARGV\n"' Foo Bar Bletch
Foo Bar Bletch
```

Suppose I have a file that contains a list of files, one filename per line, that I want to manipulate. I can see the file names when I list the files in code listing 12.

Code Listing 12: The filenames in files.txt

```
prompt$ cat files.txt
afile1
afile2
afile3
```

The unix utility xargs can transpose its standard input into arguments for another command. I want to take this list of filenames and make them the arguments of the wc command so I can count the number of lines in each file. In code listing 13 the xargs command takes its standard input, the list of filenames, and makes them the arguments for wc.

Code Listing 13: Count lines in files

```
prompt$ cat files.txt | xargs wc -l
          54 afile1
          54 afile2
          54 afile3
         162 total
```

Code listing 13 is the same as if I typed this directly, as in code listing 14.

Code Listing 14: Count lines in files

```
prompt$ wc -l afile1 afile2 afile3
```

2.3 Playing with find

In code listing 15 I reimplement the find command option “-type d” using a perl one-liner and xargs. The find command recursively outputs a list of filenames starting from a specified directory and matching certain criteria. In this case, the criteria, “-type d”, lists only directories.

Code Listing 15: Using find

```
prompt$ find . | xargs perl -e '@ARGV = grep( -d $_ , @ARGV); print "@ARGV"'
```

The xargs command takes the list of filenames and makes them the arguments to the one-liner in code listing 15. The one-liner then uses a grep expression to filter @ARGV for filenames that are directories using the -d file test operator and then prints the results.

3 Useful command line switches

Perl command line options shorten one-liners by adding automatic processing to the small script I create using the -e option. Perl has many other useful options besides the ones I show. See the perlrun manual page for the details.

3.1 The -e switch

The perl interpreter takes each -e argument as a fragment of Perl code and executes it. Each -e switch on the command line is taken as a line in a script. If I paste the contents of each -e switch into a file, and run Perl on that file, I have the exact same effect as the -e switch. Code listing 16 rewrites code listing 1 with two -e switches.

Code Listing 16: Multiple -e switches

```
prompt$ perl -e 'print "Hello ";' -e 'print "world\n";'
Hello world
```

Each code bit (in the outer single quotes) is a single string that the shell parses as a separate token, so the shell sees the four tokens in code listing 17.

Code Listing 17: Multiple -e switches, as tokens

```
-e
print "Hello ";
-e
print "world\n";
```

3.2 The -n switch

The -n switch wraps a while loop around your program. In code listing 18, the loop reads lines of input with the diamond operator, sets $_ to the contents of each line, then executes the code bits I specify with the -e switch.

Code Listing 18: Using -n

```
while (<>) {
  <-e argument>
  <-e argument>
}
```

In code listing 19 I create my own cat program.

Code Listing 19: Reimplementing cat

```
        prompt$ perl -ne 'print $_' afile
```

3.3 The -p switch

The -p switch does the same thing and prints the value of $_ at the end of each iteration.

Code Listing 20: Using -p

```
while (<>) {
  <-e argument>
  <-e argument>
  print;
}
```

In code listing 20 the loop reads lines in standard input, sets $_ to the contents of each line, executes the -e args, and then prints $_. I can use this to modify lines from an output listing.

For example, I can remove the permissions column on a "ls -l" output file listing. In code listing 21 I substitute the first group of non-whitespace and the space after it with nothing.

Code Listing 21: Remove the first column

```
prompt$ ls -l | perl -pe 's/\S+ //'
```

3.4 Using modules

I can use modules on the command line with the -M switch. The -M<module> switch is the equivalent of including "use <module>;" in the virtual scripts I create. In code listing 22, I use the IO::Handle module to set the standard output autoflush option.

Code Listing 22: Using modules

```
prompt$ cat afile | perl -MIO::Handle -e 'STDOUT->autoflush(1); while (<>) { print }'
```

Normally, I use strict and turn on warnings in my scripts and I can do this in one-liners as well. In code listing 23 I include the strict module with -Mstrict, and turn on warnings by adding -w.

Code Listing 23: Using strict

```
prompt$ cat afile | perl -w -Mstrict -e 'my $var = 17; print $var'
```

If I do not declare $var, the strict module catches it as it does in code listing 24.

Code Listing 24: Undeclared variables

```
prompt$ cat afile | perl -w -Mstrict -e '$var = 17; print $var'
Global symbol "$var" requires explicit package name at -e line 1.
Execution of -e aborted due to compilation errors.
```

In code listing 25, Perl warns about $var which I used without initializing it.

Code Listing 25: Uninitialized variables

```
cat afile | perl -w -Mstrict -e 'my $var; print $var'
Use of uninitialized value at -e line 1.
```

4 Wrestling with the shell

Quote marks, double and single, as well as the dollar sign, are part of the shell syntax. If I need to use these characters in my string, I must escape them. Each shell has a slightly different syntax for its special characters, and different platforms may handle escaping them differently. Code listing 26 shows several examples of escaping shell metacharacters.

Code Listing 26: Escaping shell metacharacters

```
prompt$ echo "the variable \$USER is "\""$USER"\"" "
the variable $USER is "jbay"
```

I can do several things to avoid shell quoting problems. In code listing 27, the program outputs a malformed SQL statement because the literal a3 is not quoted. The single quotes disappear because I used single quotes for my -e code bit, but I need quotes around 'a3' so the SQL parser knows that a3 is a literal string and not a column name.

Code Listing 27: Misquoted SQL

```
prompt$ perl -e 'print "select * from foo where bar='a3'\n"'
select * from foo where bar = a3
```

In code lisitng 28 I use Perl's chr() function to add any literal character (including quotes) using its ordinal ascii value. I can concatenate chr(39), the single quote, with the rest of the SQL string.

Code Listing 28: Using chr() to get literal values

```
prompt$ perl -e 'print "select * from foo where bar=" . chr(39) . "a3" . chr(39) . "\n"'
select * from foo where bar='a3'
```

In code listing 29 I use the generalized quote operators, q and qq, instead of single and double quotes. I can use single or double ticks inside the resulting perl string because they are no longer delimiters.

Code Listing 29: Generalized quotes

```
prompt$ perl -e 'print qq#select * from foo where bar="a3"\n#'
select * from foo where bar="a3"
```

Code listing 30 uses the back-whack character,"\", to escape the quote marks, but the syntax is rather unwieldy – `'\''`. In most shells, I have to close the prior string with the first tick, then put in the literal tick with a back-whack , and then start the next string with a third tick. The shell then concatenates those strings into a single string before it executes them.

Code Listing 30: Escaping quote characters

```
prompt$ perl -e 'print "select count(*) from foo where bar ='\''a3'\''\n"'
select * from foo where bar ='a3'
```

5 Start and End tricks

I can execute code before or after my -e program with the BEGIN or END keyword respectively. The END block in code listing 31 prints the sum after the while loop finishes.

Code Listing 31: END block

```
ls -lAF | perl -ne 'next if /^d/; $sum += (split)[4]; END{ print "$sum\n" }'
```

In code listing 32 the BEGIN executes its block before an implicit loop starts. I can initialize the variable $sum to 1024 before the loop begins if I use a BEGIN block.

Code Listing 32: BEGIN block

```
ls -lAF | perl -ne 'BEGIN{$sum=1024} next if /^d/; $sum += (split)[4]; END{ print "$sum\n" }'
```

6 References

Chapter 6, “Social Engineering, Cooperating with Command Interpreters”, Programming Perl - Larry Wall, Tom Christiansen, & Jon Orwant.

The following perl manual pages come with the standard Perl distribution and can be found online at Perldoc.com, http://www.perldoc.com, or from the command line with “perldoc pagename”.

- perlrun - perl interpreter options
- perlfaq3 “Why don’t Perl one-liners work on my DOS/Mac/VMS system?”

The following unix manual pages may be found online at several sites, including http://www.bsdi.com/bsdi-man/, or from the command line with “man pagename”.

- find
- wc
- xargs