Efficient & effective Perl Pipelines

Simon Prochnik with code from Scott Cain

Getting Help, Learning more Perl

You can use `perldoc` to get help on a perl topic or module and `perldoc -f` to get help on a perl function. For example

```
% perldoc perlref
PERLREF(1)            User Contributed Perl Documentation           PERLREF(1)

NAME
       perlref - Perl references and nested data structures

NOTE
       This is complete documentation about all aspects of references.  For a
       shorter, tutorial introduction to just the essential features, see
       perlreftut.

DESCRIPTION
       Before release 5 of Perl it was difficult to represent complex data
       structures, because all references had to be symbolic--and even then it
       was difficult to refer to a variable instead of a symbol table entry.
       Perl now not only makes it easier to use symbolic references to
...
```

```
% perldoc File::Basename

System::Library::Perl:User8ContrSystem::Library::Perl::5.18::File::Basename(3)

NAME
       File::Basename - Parse file paths into directory, filename and suffix.

SYNOPSIS
           use File::Basename;

           ($name,$path,$suffix) = fileparse($fullname,@suffixlist);
           $name = fileparse($fullname,@suffixlist);

           $basename = basename($fullname,@suffixlist);
           $dirname  = dirname($fullname);

DESCRIPTION
       These routines allow you to parse file paths into their directory,
       filename and suffix.
...
```

```
% perldoc -f split
F_EQYSRHWO(1)         User Contributed Perl Documentation        F_EQYSRHWO(1)

       split /PATTERN/,EXPR,LIMIT
       split /PATTERN/,EXPR
       split /PATTERN/
       split   Splits the string EXPR into a list of strings and returns the
               list in list context, or the size of the list in scalar
               context.
...
```

You can also get online help from
http://perldoc.perl.org/
http://perldoc.perl.org/functions/
Great introductory tutorials on lots of topics: regular expressions, debugger, objects
http://perldoc.perl.org/index-tutorials.html

Debugger

Run with a script, pass command line parameters in as normal

```
% perl -d sum.pl 4 5

Loading DB routines from perl5db.pl version 1.39_10
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(sum.pl:4):   my $a = shift @_;
  DB<1>
```

You don't need a script to enter an interactive perl debugger session

```
% perl -de 4
Loading DB routines from perl5db.pl version 1.39_10
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(-e:1):   4
  DB<1> $a = 5

  DB<2> p $a
5
  DB<3> $b = "hello world\n"

  DB<4> p $b
hello world

  DB<5> @a = qw(red green yellow blue)

  DB<6> x @a
0  'red'
1  'green'
2  'yellow'
3  'blue'
  DB<7> $ref = {red => [5,0,0],green=> [0,3,0],blue=>[0,0,6]}

  DB<8> x $ref
0  HASH(0x7f8a85063898)
   'blue' => ARRAY(0x7f8a85044f20)
      0  0
      1  0
      2  6
   'green' => ARRAY(0x7f8a85007fc0)
      0  0
      1  3
      2  0
   'red' => ARRAY(0x7f8a843a9a30)
      0  5
      1  0
      2  0
  DB<9>
```

Type 'h' to get help in the debugger

```
  DB<10> h
List/search source lines:               Control script execution:
  l [ln|sub]  List source code            T           Stack trace
  - or .      List previous/current line  s [expr]    Single step [in expr]
  v [line]    View around line            n [expr]    Next, steps over subs
  f filename  View source in file         <CR/Enter>  Repeat last n or s
  /pattern/ ?patt?   Search forw/backw    r           Return from subroutine
  M           Show module versions        c [ln|sub]  Continue until position
Debugger controls:                        L           List break/watch/actions
  o [...]     Set debugger options        t [n] [expr] Toggle trace [max depth]
  <[<]|{[{]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cnd] Set breakpoint
  ! [N|pat]   Redo a previous command     B ln|*      Delete a/all breakpoints
  H [-num]    Display last num commands   a [ln] cmd  Do cmd before line
  = [a val]   Define/list an alias        A ln|*      Delete a/all actions
  h [db_cmd]  Get help on command         w expr      Add a watch expression
  h h         Complete help page          W expr|*    Delete a/all watch exprs
  |[|]db_cmd  Send output to pager        ![!] syscmd Run cmd in a subprocess
  q or ^D     Quit                        R           Attempt a restart
Data Examination:     expr     Execute perl code, also see: s,n,t expr
  x|m expr       Evals expr in list context, dumps the result or lists methods.
  p expr         Print expression (uses script's current package).
  S [[!]pat]     List subroutine names [not] matching pattern
  V [Pk [Vars]]  List Variables in Package.  Vars can be ~pattern or !pattern.
  X [Vars]       Same as "V current_package [Vars]".  i class inheritance tree.
  y [n [Vars]]   List lexicals in higher scope <n>.  Vars same as V.
  e     Display thread id     E Display all thread ids.
For more help, type h cmd_letter, or run man perldebug for all docs.
```

One-line perl

```
% perl -e 'print 5+4,"\n"'
9
```

Is a module installed? Yes if there is no error.

```
% perl -e 'use File::Basename'
%
```

Install with this if you have permission

```
% sudo cpan
```

or this if you don't

```
% cpan
cpan> install Bio::AlignIO::clustalw
```

Reformat files on the fly with perl one-liners

This uses some tricks we have deliberately avoided in the class, including the dreaded implicit `$_`. It's so useful for more advanced bioinformatics work that I include it here, but until it's really material that you should work up to on your own.

Here's a fasta file, volvox_seq.fa. How can we extract the IDs and print them in a single column?

```
>vca4886446_93762
MSPPPTHSTTESRMAPPSQSSTPSGDVDGS
>vca4887371_120236
MAGLHSVPKLSARRPDWELPELHGDLQLAP
>vca4887497_89954
MAYKLFGTAAVLNYDLPAERRAELDAMSME
>vca4888938_93984
MLHTDLQPPRCRTSGPRPDPLRMETRARER
```

We can use two perl command line switches `-ne` to make a mini perl script that we write on the command line, or a 'perl one-liner'. All the switches are described in Chapter 17 of Programming Perl. Here's the syntax
`perl -ne '<script>' [files...]`

which writes the loop below around `<script>` like so.

```
#!/usr/bin/perl
#no use warnings or strict in this example
foreach $file (@_) {
    open(IN,'<', $file);
    while (<IN>) {
        # your script goes here
    }
}
```

So you can write this script in short hand

```
#!/usr/bin/perl
#no use warnings or strict in this example
foreach $file (@_) {
    open(IN,'<', $file);
    while (<IN>) {
        # your script goes here
        if ($_ =~ /^>(\S+)/) {
            print "$1\n";
        }
    }
}
```

```
% perl -ne 'if (/^>(\S+)/) {print "$1\n"}' volvox_AP2EREBP.fa
vca4886446_93762
vca4887371_120236
vca4887497_89954
```

Very often used with pipes, eg to print duplicate IDs across multiple fasta files

```
% perl -ne 'if (/^>(\S+)/) {print "$1\n"}' *.fa | sort | uniq -d
```

Constructing a simple pipeline

Let’s assume we have a multiple fasta file and we want to use perl to run the program clustalw to make a multiple sequence alignment and read in the results. Here is the pipeline:

1. get fasta seq filename
2. construct output filename
3. generate command line that will align sequences with clustalw,
4. read in/parse output file,
5. (do something with the data)

We haven't covered part 3 in the course yet. We need to figure out how to run clustalw on the command line. Then we can use `system()` in perl to run a command as if we typed it in. It returns the exit value of the command. 0 (false) means success.
Any other number (true) is the error code for what went wrong.

```
my $result = system(“ls -ltr”);
print “exit value is $result\n”;
```

output is

```
drwx------+  3 admin  staff     102 Sep 26 17:02 Pictures
drwx------+  3 admin  staff     102 Sep 26 17:02 Music
drwx------+  3 admin  staff     102 Sep 26 17:02 Movies
drwxr-xr-x+  5 admin  staff     170 Sep 26 17:02 Public
drwx------+  4 admin  staff     136 Sep 26 17:05 Documents
drwx------@ 48 admin  staff    1632 Oct 17 08:47 Library
drwx------+  9 admin  staff     306 Oct 17 08:49 Desktop
drwx------+  5 admin  staff     170 Oct 17 14:46 Downloads
exit value is 0
```

Here's an example where the file we try to list doesn't exist

```
my $result = system("ls this_file_is_missing.txt”);
print “exit value is $result\n”;
```

output is

```
ls: this_file_is_missing.txt: No such file or directory
exit value is 256
```

Start looking for help running clustalw

A google search for 'clustalw command line' returns this

```
USE OF OPTIONS
     All parameters of Clustalw can be used as options
with a "-" That permits to use Clustalw in a script or
in batch.
    $ clustalw -options
     CLUSTAL W (1.7) Multiple Sequence Alignments
    clustalw option list:-
             -help
                -options
                -infile=filename
                -outfile=filename
                -type=protein OR dna
                -output=gcg OR gde OR pir OR phylip
```

So our command might look like this

```
% clustalw -infile=ExDNA.fasta -outfile=ExDNA.aln -type=dna
```

Default output format is clustalw format (*.aln) Run it to test. Did it do *exactly* what you want/expect?

Command line Running a command and handling failure clustalw -infile=ExDNA.fasta -outfile=ExDNA.aln -type=dna Script

```
#!/usr/bin/perl
use strict;
use warnings;
my $file = ‘ExDNA.fasta’;
my $clustFile = ‘ExDNA.aln’;
# build command line
my $cmd = “clustalw -infile=$file -outfile=$clustFile -type=dna”;
print “Call to clustalw $cmd\n”; # show command
my $oops = system $cmd; # system call and save return
                        # value in $oops
die “FAILED $!” if $oops; # $oops true if failed
```

Let's put the system... die code in a sub() inside a module as we'll be using it a lot.

```
# file is Util.pm
package Util;
use strict;
our @EXPORT = qw(do_or_die);
use Exporter;
use base 'Exporter';
  # allow do_or_die() to be exported
# without specifying
# Util::do_or_die()
# --------------------------------------------------------------
sub do_or_die {
  my $cmd = shift;
  print "CMD: $cmd\n";
  my $oops = system $cmd;
  die "Failed" if $oops;
}
# --------------------------------------------------------------
1;
```

Here's our script updated to use the do*or*die() subroutine from the Util package

```
#!/usr/bin/perl
use strict;
use warnings;
use lib '/pfbhome/sprochnik/lib'; # you might need to tell perl where to find Ut
               # or with something like this
use Util;

my $file = 'ExDNA.fasta';
my $clustFile = 'ExDNA.aln';

# build command line
my $cmd = "clustalw -infile=$file -outfile=$clustFile -type=dna";

do_or_die($cmd); # I use this all the time
```

Next we need to figure out how to parse the alignment file. This means reading the contents into memory. Bioperl can read almost every bioinformatics file format. Look in bioperl documentation for help. See HOWTOs http://www.bioperl.org/wiki/HOWTOs

> Alt

Then go to the AlignIO tutorial which looks something like this, with a great example snippet of code you can copy and paste and edit.

> ## Abstract
>
> This is a HOWTO that talks about using Bio::AlignIO and Bio::SimpleAlign) to create and analyze alignments. It also discusses how to run various applications that create alignment files. See the list of alignment formats for details on each format.
>
> ## AlignIO
>
> Data files storing multiple sequence alignments appear in varied formats and is the Bioperl object for conversion of alignment files. AlignIO is patterned on the object and its commands have many of the same names as the commands in . Just as in the object can be created with -file and -format options:
>
> ```
> use Bio::AlignIO; my $io = Bio::AlignIO->new(
>                         -file => "receptors.aln",
>                         -format => "clustalw" );
> ```
>
> If the -format argument isn't used then Bioperl will try and determine the format based on the file's suffix, in a case-insensitive manner.

> ...
>
> The basic syntax for AlignIO is similar to that of SeqIO:
>
> ```
> use Bio::AlignIO;
>
> $in = Bio::AlignIO->new(-file => "inputfilename" ,
>                        -format => 'fasta');
>
> $out = Bio::AlignIO->new(-file => ">outputfilename",
>                         -format => 'pfam');
>
> while ( my $aln = $in->next_aln ) {
>    $out->write_aln($aln);
> }
> ```
>
> The returned object, $aln, is a Bio::SimpleAlign object rather than a Bio::Seq object.
>
> Bio::AlignIO also supports the tied filehandle syntax described for Bio::SeqIO.
>
> ...

Let's add the new piece to our script.

```
#!/usr/bin/perl
use strict;
use warnings;

use Util;
use Bio::AlignIO;

my $file = ‘ExDNA.fasta’;
my $clustFile = ‘ExDNA.aln’;
# build command line
my $cmd = “clustalw -infile=$file -outfile=$clustFile -type=dna”;
#run command
do_or_die($cmd);
# parse in alignment file
my $in = Bio::AlignIO->new(-file => $clustFile,
                         -format => 'clustalw');
while ( my $aln = $in->next_aln() ) {
...
}
```

See bioperl-run for bioperl modules that can run programs like clustalw

Key perl coding style points

- use strict;
- use warnings;
- Put all the hard stuff in subroutines so you can write clean subroutine calls.
- If you want to re-use a subroutine several times, put it in a module and re-use the module e.g. Util.pm
- #comments (ESC-; makes a comment in EMACS) • comment what a subroutine expects and returns
- comment anything new to you or unusual
- Use the correct amount of indentation for loops, logic and subroutines
- coding time = thinking/design (10%) + code writing (30%) + testing and debugging (60%)
- Re-use and modify existing code as much as possible
- Write your code in small pieces and test each piece as you go. Debugger? • Test with a small (fake?) data file. Debugger?
- Use more complicated tools/code only if you need to
- Think about the big picture:
- total time = coding time + run time + analysis time + writing up results
- will speeding up your code take longer than waiting for it to complete? Your time is valuable
- Check your input data and your output data • Check it again.
- are there unexpected characters, line returns (\r or \n ?), whitespace at the end of lines, spaces instead of tabs.You can use

```
% od -c mydatafile | more
```

- are there missing fields/characters, duplicated IDs?
- what about header lines, comments, corrupt data?
- Is the output exactly what you expect?