Designing and Writing Perl Pipelines

Using perl as bioinformatics glue

Simon Prochnik
with code from Scott Cain

Built-in perldoc <perl topic> to get help

```
% 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
       ...
```

Also available online at http://perldoc.perl.org/index-tutorials.html

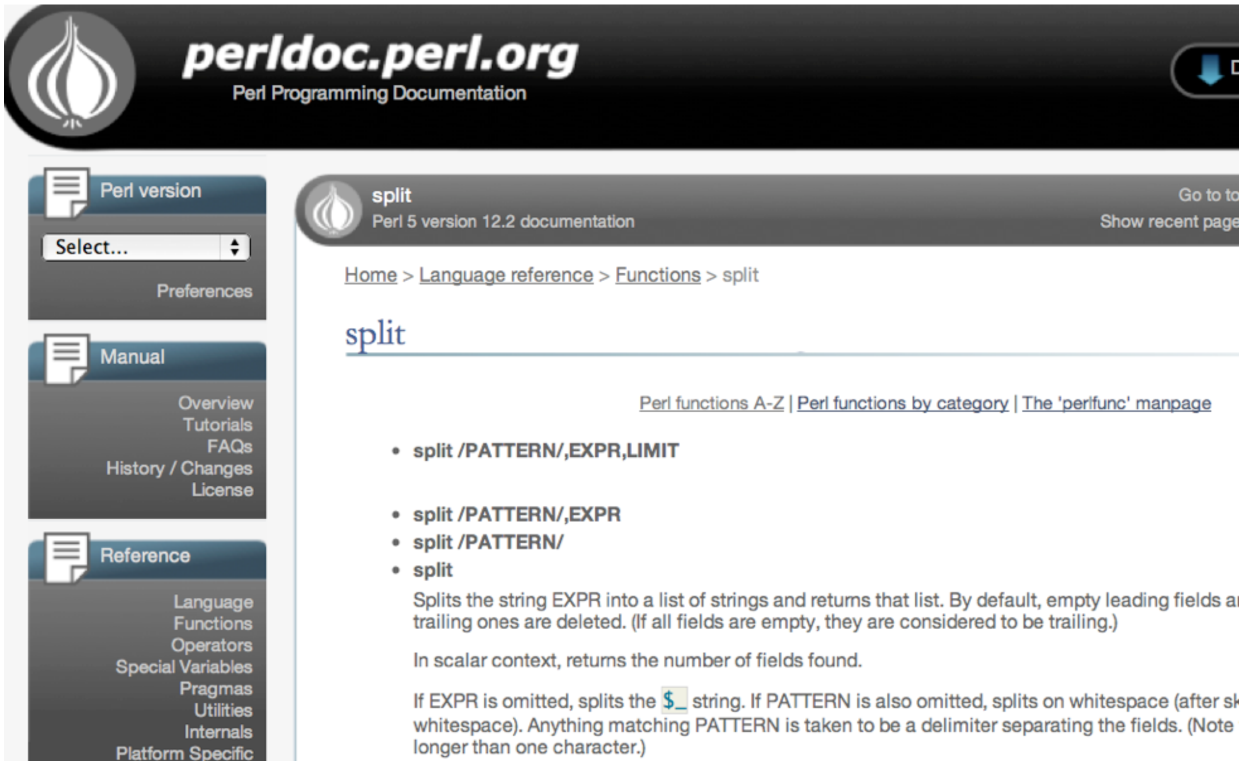
Built-in perldoc -f <command> to get help

```
% perldoc -f split

       split /PATTERN/,EXPR,LIMIT
       split /PATTERN/,EXPR
       split /PATTERN/
       split   Splits the string EXPR into a list of strings and returns that
               list.  By default, empty leading fields are preserved, and
               empty trailing ones are deleted.  (If all fields are empty,
               they are considered to be trailing.)
```

Get online help from perldoc.perl.org

http://perldoc.perl.org/functions/split.html



split

Perl functions A-Z | Perl functions by category | The 'perlfunc' manpage

- **split /PATTERN/,EXPR,LIMIT**
- **split /PATTERN/,EXPR**
- **split /PATTERN/**
- **split**

  Splits the string EXPR into a list of strings and returns that list. By default, empty leading fields a
  trailing ones are deleted. (If all fields are empty, they are considered to be trailing.)

  In scalar context, returns the number of fields found.

  If EXPR is omitted, splits the $_ string. If PATTERN is also omitted, splits on whitespace (after sk
  whitespace). Anything matching PATTERN is taken to be a delimiter separating the fields. (Note
  longer than one character.)

Running your script in the perl debugger

```
> perl -d myScript.pl
Loading DB routines from perl5db.pl version 1.28
Editor support available.
Enter h or `h h' for help, or `man perldebug' for more help.
main::(myScript.pl:3): print "hello world\n";
  DB<1>
```

`h`	help
`q`	quit
`n or s`	next line or step through next line
`<return>`	repeat last n or s
`!`	repeat last command
`c 45`	continue to line 45
`b 45`	break at line 45
`b 45 $a == 0`	break at line 45 if $a equals 0
`p $a`	print the value of $a
`x $a`	unpack or extract the data structure in $a
`R`	restart the script

Exploring data structures with the debugger

```
> perl -de 4
Loading DB routines from perl5db.pl version 1.28
Editor support available.

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

main::(-e:1):4
  DB<1> $a = {foo => [1,2] , boo => [2,3] , moo => [6,7]}
  DB<2> x $a
0  HASH(0x8cd314)
   'boo' => ARRAY(0x8c3298)
      0   2
      1   3
   'foo' => ARRAY(0x8d10d4)
      0   1
      1   2
   'moo' => ARRAY(0x815a88)
      0   6
      1   7
```

More perl tricks: one line perl

```
> perl -e <COMMAND>

> perl -e '@a = (1,2,3,4);print join("\t",@a),"\n"'
1     2     3    4


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

- see Chapter 19, p. 492-502 Perl book 3rd ed.

Contents of fasta file volvox_AP2EREBP.fa

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

Is a module installled?

one-line perl program with '-e'

this is the program in quotes

```
% perl -e 'use Bio::AlignIO::clustalw'
```

all ok: no errors

The module in the next example hasn't been installed (it doesn't actually exist)

```
% perl -e 'use Bio::AlignIO::myformat'
Can't locate Bio/AlignIO/myformat.pm in
@INC (@INC contains: /sw/lib/perl5 /sw/
lib/perl5/darwin /Users/simonp/lib /
Users/simonp/Library/Perl/5.8.1/darwin-
thread-multi-2level /Users/simonp/
Library/Perl/5.8.1 /Users/simonp/com_lib
/Users/simonp/cvs/bdgp/software/perl-
modules ...
```

perl can't find the module in any of the paths in the PERL5LIB list (which is in the perl variable @INC).
You can add directories with
use lib '/Users/yourname/lib';
after the use strict; at the beginning of your script.

To install a module

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

Toy example: Finding out how to run a small task

- 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 are some sequences in fasta format

```
>vca4886446_93762
ACGCTAGCGAGCCTAGAGCTTATTTTATGC
>vca4887371_120236
ACGCTACCGAACCTAGTGCTTAAATTATGC
>vca4887497_89954
ACGCTAGCGAACCTAGTGCTTAAATAATCC
>vca4888938_93984
ACGCTAGGGTACCTTGTGCTTAAATAAACC
```

Here is the pipeline:
get fasta seq filename,
construct output filename,
generate command line that will align sequences with clustalw,
read in/parse output file,
(do something with the data)

How do we start on this? -- Looking for help

- Google
  - <program name> documentation / docs / command line
  - e.g. google 'clustal command line'

```
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
```

Build a command line from the options you need **and test it out**

```
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
```

Command line would be:
```
% clustalw -infile=ExDNA.fasta -outfile=ExDNA.aln -type=dna
```
Default output format is clustalw format (*.aln)
Did it do **exactly** what you want/expect when you tested it?

Running a command with `system( )`

The perl command `system` runs a unix command.
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
```
bin
bowtie_test
data
my_file.pl
exit value is 0
```

```
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
```

system( ) tells you if the command ran ok

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

my $file = shift;
my $options = "-ltrh";
my $command = 'ls';

my $result = system("$command $options
$file");
print "exit status is $result\n";
die "Failed!\n"  if $result;
```

```
% perl system.pl ‘*.pl'
-rw-r--r--   1 simonp  admin   413B Oct 20 13:33 ref_perl.pl
-rw-r--r--   1 simonp  admin   393B Oct 20 13:51 ref_no_sub.pl
-rw-r--r--   1 simonp  admin   189B Oct 20 14:51 system.pl
exit status is 0
```

Always check the exit status for success or errors

Running a command and handling failure

Command line

```
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
```

Util.pm package for nice reusable utility functions

```
package Util;
use strict;
our @EXPORT = qw(do_or_die);    # allow do_or_die() to be exported
                               # without specifying
                               # Util::do_or_die()
use Exporter;
use base 'Exporter';


# -------------------------------------------------------------
sub do_or_die {
  my $cmd = shift;
  print "CMD: $cmd\n";
  my $oops = system $cmd;
  die "Failed" if $oops;
}
# -------------------------------------------------------------

1;
```

Util.pm in a script

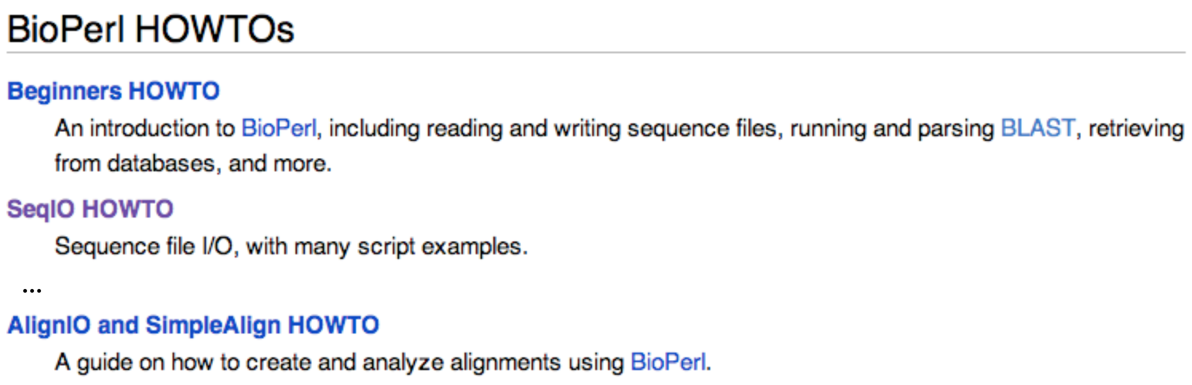
```
#!/usr/bin/perl
use strict; use warnings;
use lib ‘lib’; # you might need to tell perl where to find Util.pm
               # or with something like this
               # use lib ‘/Users/simonp/lib’;
use Util;


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”;       # don’t need this
#  anymore because do_or_die shows the command

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

Next step: How do we find out how to parse the clustalw alignment file (without even knowing what the file format is)?

The output is a clustalw multiple sequence alignment in the file ExDNA.aln
Look in bioperl documentation for help.
See HOWTOs
http://www.bioperl.org/wiki/HOWTOs



BioPerl HOWTOs

Beginners HOWTO

An introduction to BioPerl, including reading and writing sequence files, running and parsing BLAST, retrieving from databases, and more.

SeqIO HOWTO

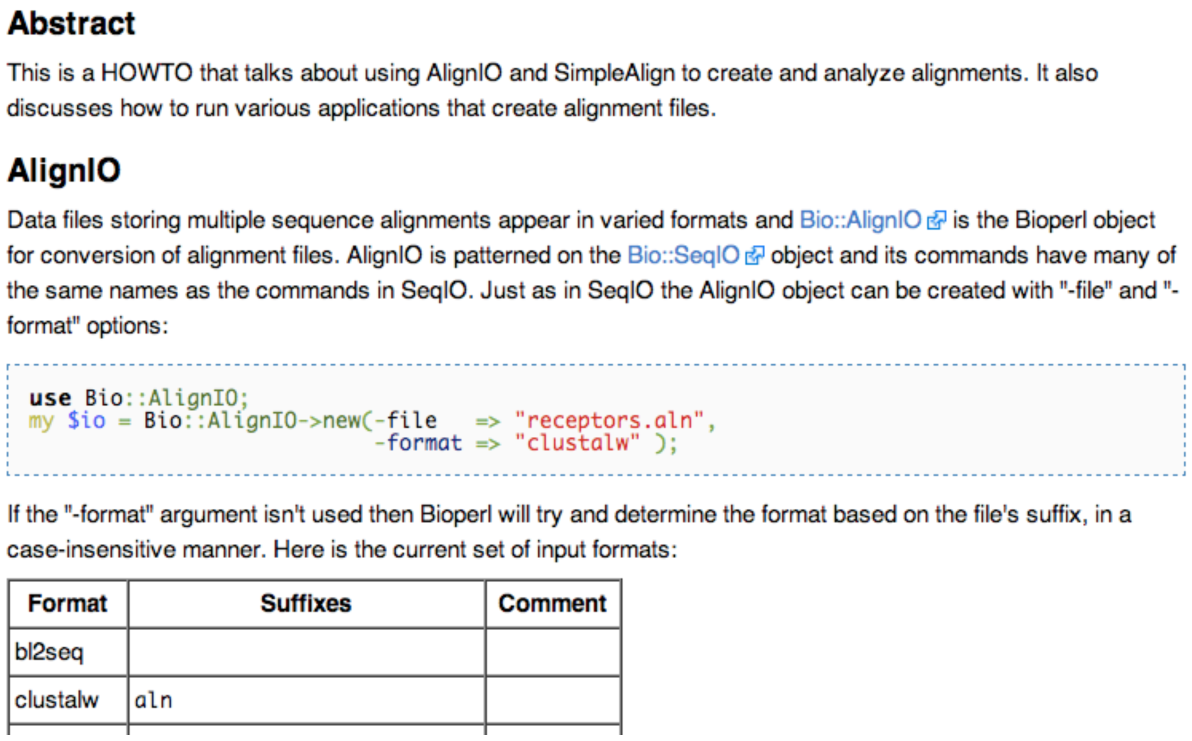
Sequence file I/O, with many script examples.

...

AlignIO and SimpleAlign HOWTO

A guide on how to create and analyze alignments using BioPerl.

Help on AlignIO from bioperl



Abstract

This is a HOWTO that talks about using AlignIO and SimpleAlign to create and analyze alignments. It also discusses how to run various applications that create alignment files.

AlignIO

Data files storing multiple sequence alignments appear in varied formats and Bio::AlignIO is the Bioperl object for conversion of alignment files. AlignIO is patterned on the Bio::SeqIO object and its commands have many of the same names as the commands in SeqIO. Just as in SeqIO the AlignIO object can be created with "-file" and "-format" options:

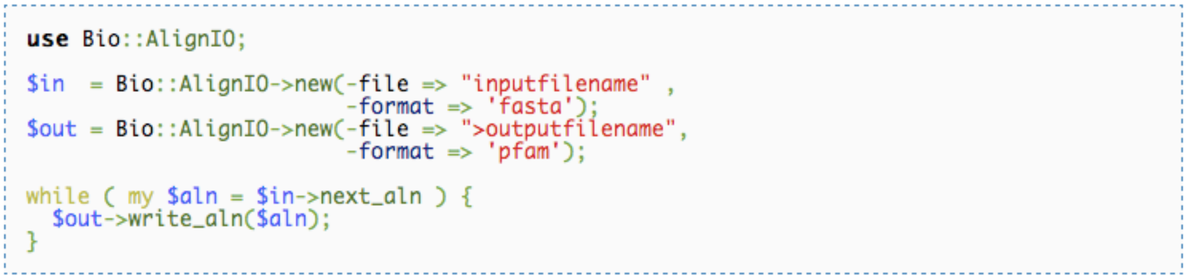
```
use Bio::AlignIO;
my $io = Bio::AlignIO->new(-file   => "receptors.aln",
                           -format => "clustalw" );
```

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. Here is the current set of input formats:

Format	Suffixes	Comment
bl2seq		
clustalw	aln	

More help on AlignIO from bioperl

Here’s a more useful synopsis



```
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);
}
```

Let’s add this to our script

Use bioperl to parse the clustalw alignment

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

Script
```
#!/usr/bin/perl
use strict; use warnings;
use Util;
use Bio::AlignIO;
my $file = ‘ExDNA.fasta’;
my $clustFile = ‘ExDNA.aln’;
my $cmd = “clustalw -infile=$file -outfile=$clustFile
 -type=dna”;                    # build command line
do_or_die($cmd);
my $in  = Bio::AlignIO->new(-file   => $clustFile,
                         -format => 'clustalw');
while ( my $aln = $in->next_aln() ) {
       ...
    }
```

We just wrote a script to parse in a clustalw alignment without having to worry about the file format

- That's the point of bioperl and object-oriented programming.
- You don't need to know the details of the file format to be able to work with it or how the alignment is stored in memory.
- Here's a sample protein sequence alignment in case you are curious

```
CLUSTAL W (1.74) multiple sequence alignment

 seq1 -----------------------KSKERYKDENGGNYFQLREDWWDANRETVWKAITCNA
 seq2 ---------------YEGLTTANGXKEYYQDKNGGNFFKLREDWWTANRETVWKAITCGA
 seq3 ----KRIYKKIFKEIHSGLSTKNGVKDRYQN-DGDNYFQLREDWWTANRSTVWKALTCSD
 seq4 ------------------------SQRHYKD-DGGNYFQLREDWWTANRHTVWEAITCSA
 seq5 --------------------NVAALKTRYEK-DGQNFYQLREDWWTANRATIWEAITCSA
 seq6 ------FSKNIX--QIEELQDEWLLEARYKD--TDNYYELREHWWTENRHTVWEALTCEA
 seq7 --------------------------------------------------KELWEALTCSR

 seq1 --GGGKYFRNTCDG--GQNPTETQNNCRCIG----------ATVPTYFDYVPQYLRWSDE
 seq2 P-GDASYFHATCDSGDGRGGAQAPHKCRCDG---------ANVVPTYFDYVPQFLRWPEE
 seq3 KLSNASYFRATC--SDGQSGAQANNYCRCNGDKPDDDKP-NTDPPTYFDYVPQYLRWSEE
 seq4 DKGNA-YFRRTCNSADGKSQSQARNQCRC---KDENGKN-ADQVPTYFDYVPQYLRWSEE
 seq5 DKGNA-YFRATCNSADGKSQSQARNQCRC---KDENGXN-ADQVPTYFDYVPQYLRWSEE
 seq6 P-GNAQYFRNACS----EGKTATKGKCRCISGDP----------PTYFDYVPQYLRWSEE
 seq7 P-KGANYFVYKLD-----RPKFSSDRCGHNYNGDP---------LTNLDYVPQYLRWSDE
```

bioperl-run can run clustalw and many other programs

- The Run package (bioperl-run) provides wrappers for executing some 60 common bioinformatics applications (bioperl-run in the repository system Git, see link below)
  - Bio::Tools::Run::Alignment::clustalw
- There are several pieces to bioperl, which are all listed here
- http://www.bioperl.org/wiki/Using_Git
  - bioperl-live Core modules including parsers and main objects
  - bioperl-run Wrapper modules around key applications
  - bioperl-ext Ext package has C extensions including alignment routines and link to staden IO library for sequence trace reads.
  - bioperl-pedigree
  - bioperl-microarray
  - bioperl-gui
  - bioperl-db

~~Smart~~ Essential coding practices

- `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 strategy

- 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?

Installing and using a simple perl profiler Devel::Profile

NAME

Devel::Profile - tell me why my perl program runs so slowly

SYNOPSIS

```
    perl -d:Profile program.pl
    less prof.out
```

DESCRIPTION

The Devel::Profile package is a Perl code profiler. This will collect information on the execution time of a Perl script and of the subs in that script. This information can be used to determine which subroutines are using the most time and which subroutines are being called most often.

Install with

```
sudo cpan
cpan[1]> install Devel::DProf
```

Why is my script ‘slow’? Usually this means the script could be designed better. Perl is not inherently slow.
Here are some ways to tell if your script is ‘slow’
i) it takes longer to run than it took to write or
ii) you actually notice that it is slow.

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

print "This is version 1, passes whole array to sub\n";
my $file = shift;
my @ids = read_file($file);

sub read_file {
    my $file = shift;
    my @id_list;
    open (my $fh,"<",$file);
    while (my $id = <$fh>) {
        chomp $id;
        @id_list = add($id,@id_list); # bad idea to pass a
                                      # whole array to a subroutine
    }                                 # pass a reference instead
    close($fh);
    return @id_list;
}
sub add {
# don't need to define a sub to do such a simple task
    my ($id,@list) = @_; # bad idea to do this
    push @list,$id;
    return @list; # bad idea to do this
}
```

File: pmid.10000.ids contents

```
PMID938298
PMID389187
PMID863926
PMID594930
PMID305530
PMID560705
PMID719401
. . .
```

Run the script like this.

```
% perl -d:Profile long_perl.pl pmid.10000.ids
```

The input file contains a list of 10000 pubmed PMIDs, one per line

```
% perl -d:Profile long_perl.pl pmid.10000.ids
This is version 1, passes whole array to sub
```

Devel::Profile makes a file called prof.out with information on the time perl spends in each part of the script

```
% more prof.out
time elapsed (wall):   47.4645
time running program:  47.4515  (99.97%)
time profiling (est.): 0.0131  (0.03%)
number of calls:       10006

%Time    Sec.     #calls   sec/call  F  name
58.59    27.8009       1  27.800936     main::read_file
41.40    19.6430   10000   0.001964     main::add
 0.02     0.0075       0   0.007521  *  <other>
 0.00     0.0000       1   0.000009     <anon>:long_perl.pl:2
 0.00     0.0000       1   0.000005     strict::bits
 0.00     0.0000       1   0.000004     warnings::import
 0.00     0.0000       1   0.000004     <anon>:long_perl.pl:3
 0.00     0.0000       1   0.000003     strict::import
```

Let's pass an array reference instead of an array

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

print "This is version 2, passes
an array reference to sub\n";
my $file = shift;
my @ids = read_file($file);
sub read_file {
    my $file = shift;
    my @id_list;
    open (my $fh,"<",$file);
    while (my $id = <$fh>) {
        chomp $id;
        add_ref($id,\@id_list);
    }
    close($fh);
    return @id_list;
}

sub add_ref {
    my ($id,$ref) = @_;
    push @{$ref},$id;
}
```

```
% perl -d:Profile ref_perl.pl pmid.10000.ids
This is version 2, passes an array reference to sub
[epinephrine:~] simonp% more prof.out
time elapsed (wall):   0.0587
time running program:  0.0477  (81.36%)
time profiling (est.): 0.0109  (18.64%)
number of calls:       10006

%Time  Sec.    #calls  sec/call  name
59.21  0.0283       1  0.028273  main::read_file
30.45  0.0145   10000  0.000001  main::add_ref
10.29  0.0049       0  0.004913  <other>
 0.01  0.0000       1  0.000007  <anon>:ref_perl.pl:2
```

Lastly, let's skip the sub call all together
Time spent in read_file() goes from 0.028 s down to 0.010 s, which is about three times faster

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

print "This is version 3, doesn't use a sub\n";

my $file = shift;
my @ids = read_file($file);

sub read_file {
    my $file = shift;
    my @id_list;
    open (my $fh,"<",$file);
    while (my $id = <$fh>) {
        chomp $id;
        push @id_list,$id;
    }
    close($fh);
    return @id_list;
}
```

```
time elapsed (wall):   0.0183
time running program:  0.0165  (90.47%)
time profiling (est.): 0.0017  (9.53%)
number of calls:       6

%Time  Sec.    #calls  sec/call  name
61.18  0.0101       1  0.010118  main::read_file
38.68  0.0064       0  0.006397  <other>
 0.04  0.0000       1  0.000007  <anon>:ref_no_sub.
 0.03  0.0000       1  0.000005  warnings::import
```

gene_pred_pipe.pl (by Scott Cain) part I

```
#!/usr/bin/perl -w

use strict;

use Bio::DB::GenBank;
use Bio::Tools::Run::RepeatMasker;
use Bio::Tools::Run::Genscan;
use Bio::Tools::GFF;

my $acc = $ARGV[0]; # read argument from command line

# main functions in simple subroutines
my $seq_obj = acc_to_seq_obj($acc);
my $masked_seq = repeat_mask($seq_obj);
my @predictions = run_genscan($masked_seq);
predictions_to_gff(@predictions);
warn "Done!\n";
exit(0);
#---------------------------------------
```

gene_pred_pipe.pl (by Scott Cain) part II

```
sub acc_to_seq_obj {
    #takes a genbank accession, fetches the seq from
    #genbank and returns a Bio::Seq object
    #parent script has to `use Bio::DB::Genbank`
    my $acc = shift;
    my $db  = Bio::DB::GenBank->new();
    return $db->get_Seq_by_id($acc);
}
sub repeat_mask {
    #takes a Bio::Seq object and runs RepeatMasker locally.
    #Parent script must `use Bio::Tools::Run::RepeatMasker`
    my $seq = shift;
    #BTRRM->new() takes a hash for configuration parameters
    #You'll have to set those up appropriately
    my $factory = Bio::Tools::Run::RepeatMasker->new();
    return $factory->masked_seq($seq);
}
```

gene_pred_pipe.pl (by Scott Cain) part III

```
sub run_genscan {
    #takes a Bio::Seq object and runs Genscan locally and returns
    #a list of Bio::SeqFeatureI objects
    #Parent script must `use Bio::Tools::Run::Genscan`
    my $seq = shift;
    #BTRG->new() takes a hash for configuration parameters
    #You'll have to set those up appropriately
    my $factory = Bio::Tools::Run::Genscan->new();
    #produces a list of Bio::Tools::Prediction::Gene objects
    #which inherit from Bio::SeqFeature::Gene::Transcript
    #which is a Bio::SeqFeatureI with child features
    my @genes   = $factory->run($seq);
    my @features;
    for my $gene (@genes) {
        push @features, $gene->features;
    }
    return @features;
}
sub predictions_to_gff {
    #takes a list of features and writes GFF2 to a file
    #parent script must `use Bio::Tools::GFF`
    my @features = @_;
    my $gff_out = Bio::Tools::GFF->new(-gff_version => 2,
                                       -file        => '>prediction.gff');
    $gff_out->write_feature($_) for (@features);
    return;
}
```

Getting arguments from the command line with Getopt::Long and GetOptions()

- order of arguments doesn't matter
- deals with flags, integers, decimals, strings, lists
- `myscript.pl -flag -c 4 --price 34.55 --name 'expensive flowers'`

```
use Getopt::Long;
my ($flag, $count, $price, $name);
GetOptions( "flag" => \$flag,
            "c|count=i",\$count,  # i means integer can use -c
                                  # or --count on command line
            "price=f",\$price,  # f means floating point
                                # number 0.12,3e-49
            "name=s",\$name,  # s means string
                             # NOTE: always use trailing ','
# after last element so you can add more elements later
          );
# now $flag=1, $count=4, $price = 34.55,
# and $name = 'expensive flowers'
```

genbank_to_blast.pl (by Scott Cain) part I

```
#!/usr/bin/perl -w
use strict;
use lib "/home/scott/cvs_stuff/bioperl-live";   # this will change depending
                                             #  on your machine
use Getopt::Long;
use Bio::DB::GenBank;
#use Bio::Tools::Run::RepeatMasker;   # running repeat masked first is a good
                                  # idea, but takes a while
use Bio::Tools::Run::RemoteBlast;
use Bio::SearchIO;
use Bio::SearchIO::Writer::GbrowseGFF;
use Bio::SearchIO::Writer::HTMLResultWriter;
use Data::Dumper;    # print out contents of objects etc
#take care of getting arguments
my $usage = "$0 [--html] [--gff] --accession <GB accession number>";
my ($HTML,$GFF,$ACC);
GetOptions ("html"        => \$HTML,
            "gff"         => \$GFF,
            "accession=s" => \$ACC);
unless ($ACC) {
    warn "$usage\n";
    exit(1);
}
#This will set GFF as the default if nothing is set but allowing both to be set
$GFF=1 unless $HTML; # DON'T USE ||= "I WAS TRYING TO BE CLEVER!" SAYS SCOTT
#Now do real stuff ...
```

genbank_to_blast.pl (by Scott Cain) part II

```
# Now do real stuff
# nice and neat subroutine calls
# easy to understand logic of code
my $seq_obj    = acc_to_seq_obj($ACC);
my $masked_seq = repeat_mask($seq_obj);
my $blast_res  = blast_seq($masked_seq);
gff_out($blast_res, $ACC) if $GFF;
html_out($blast_res, $ACC) if $HTML;
#------------------------------------------
```

genbank_to_blast.pl (by Scott Cain) part III

```
sub acc_to_seq_obj {
    print STDERR "Getting record from GenBank\n";
    my $acc = shift;
    my $db  = new Bio::DB::GenBank;
    return $db->get_Seq_by_id($acc);
}
sub repeat_mask {
    my $seq     = shift;
    return $seq;   #short circuiting RM since we
                   #don't have it installed, but this would be where
                   # you would run it
#    my $factory = Bio::Tools::Run::RepeatMasker-
>new();
#    return $factory->masked_seq($seq);
}
```

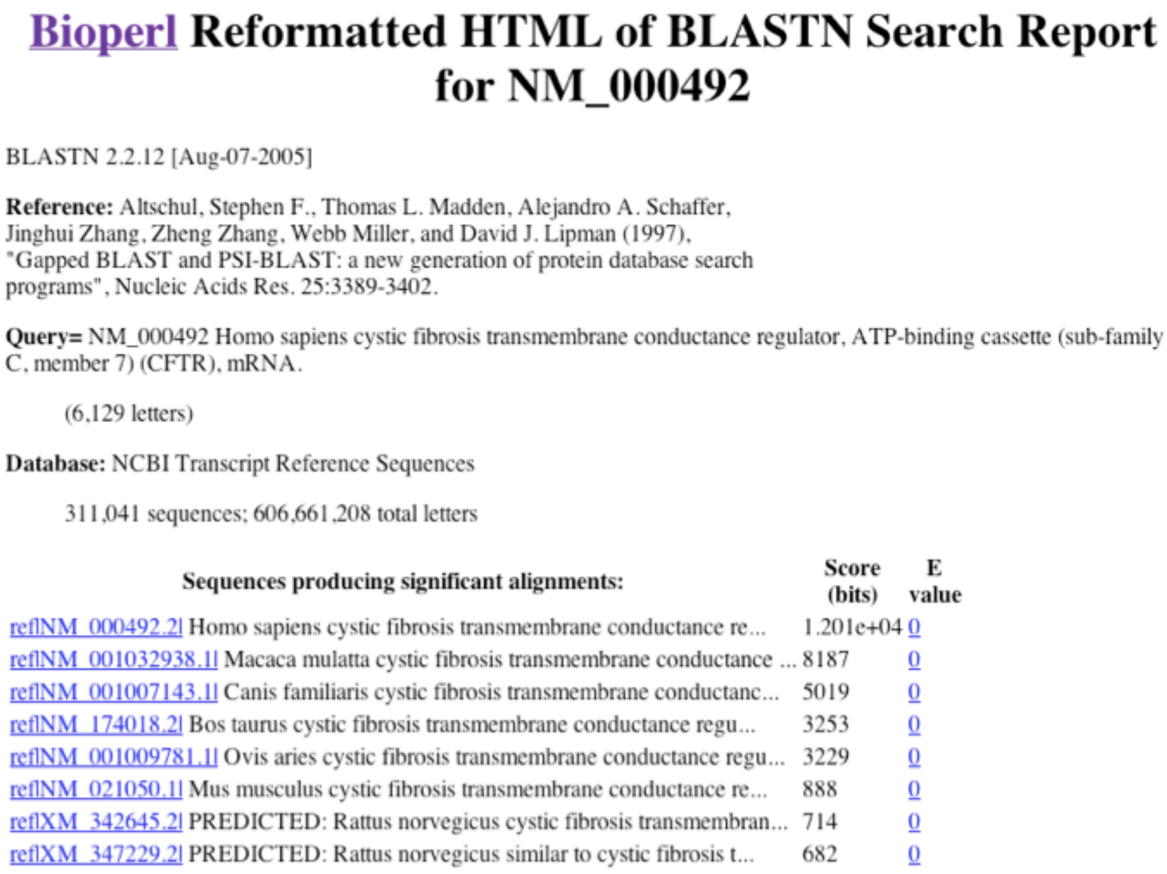
genbank_to_blast.pl (by Scott Cain) part IV

```
sub blast_seq {
    my $seq   = shift;
    my $prog  = 'blastn';
    my $e_val = '1e-10';
    my $db    = 'refseq_rna';
    my @params = (
        -prog   => $prog,
        -expect => $e_val,
        -readmethod => 'SearchIO',
        -data       => $db
    );
    my $factory = Bio::Tools::Run::RemoteBlast->new(@params);
    $factory->submit_blast($seq);
    my $v = 1; # message flag
    print STDERR "waiting for BLAST..." if ( $v > 0 );
    while ( my @rids = $factory->each_rid ) {
        foreach my $rid ( @rids ) {
            my $rc = $factory->retrieve_blast($rid);
            if( !ref($rc) ) { #waiting...
                if( $rc < 0 ) {
                    $factory->remove_rid($rid);
                }
                print STDERR "." if ( $v > 0 );
                sleep 25;
            }
            else {
                print STDERR "\n";
                return $rc->next_result();
            }
        }
    }
}
```

genbank_to_blast.pl (by Scott Cain) part V

```
sub gff_out {
    my ($result, $acc) = @_;
    my $gff_out = Bio::SearchIO->new(
        -output_format  => 'GbrowseGFF',
        -output_signif  => 1,
        -file           => ">$acc.gff",
        -reference      => 'query',
        -hsp_tag        => 'match_part',
    );
    $gff_out->write_result($result);
}
sub html_out {
    my ($result, $acc) = @_;
    my $writer   = Bio::SearchIO::Writer::HTMLResultWriter->new();
    my $html_out = Bio::SearchIO->new(
        -writer => $writer,
        -format => 'blast',
        -file   => ">$acc.html"
    );
    $html_out->write_result($result);
}
```

HTML version of blast report: NM_000492.html



Bioperl Reformatted HTML of BLASTN Search Report for NM_000492

BLASTN 2.2.12 [Aug-07-2005]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs", Nucleic Acids Res. 25:3389-3402.

Query= NM_000492 Homo sapiens cystic fibrosis transmembrane conductance regulator, ATP-binding cassette (sub-family C, member 7) (CFTR), mRNA.

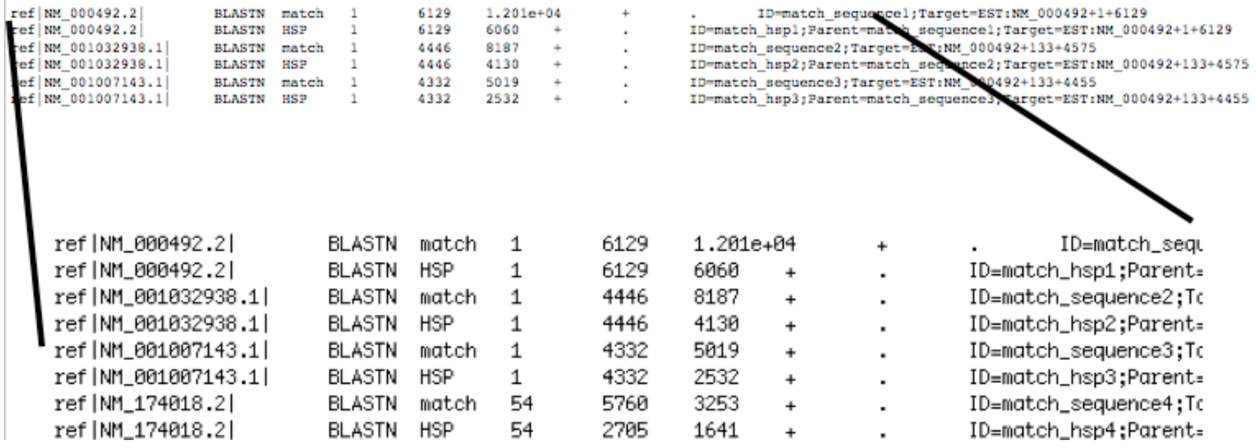
(6,129 letters)

Database: NCBI Transcript Reference Sequences

311,041 sequences; 606,661,208 total letters

Sequences producing significant alignments:	**Score (bits)**	**E value**
ref|NM_000492.2| Homo sapiens cystic fibrosis transmembrane conductance re...	1.201e+04	0
ref|NM_001032938.1| Macaca mulatta cystic fibrosis transmembrane conductance ...	8187	0
ref|NM_001007143.1| Canis familiaris cystic fibrosis transmembrane conductanc...	5019	0
ref|NM_174018.2| Bos taurus cystic fibrosis transmembrane conductance regu...	3253	0
ref|NM_001009781.1| Ovis aries cystic fibrosis transmembrane conductance regu...	3229	0
ref|NM_021050.1| Mus musculus cystic fibrosis transmembrane conductance re...	888	0
ref|XM_342645.2| PREDICTED: Rattus norvegicus cystic fibrosis transmembran...	714	0
ref|XM_347229.2| PREDICTED: Rattus norvegicus similar to cystic fibrosis t...	682	0

GFF output: NM_000492.gff



```
ref|NM_000492.2|        BLASTN  match   1       6129    1.201e+04       +       .       ID=match_sequence1;Target=EST:NM_000492+1+6129
ref|NM_000492.2|        BLASTN  HSP     1       6129    6060    +       .       ID=match_hsp1;Parent=match_sequence1;Target=EST:NM_000492+1+6129
ref|NM_001032938.1|     BLASTN  match   1       4446    8187    +       .       ID=match_sequence2;Target=EST:NM_000492+133+4575
ref|NM_001032938.1|     BLASTN  HSP     1       4446    4130    +       .       ID=match_hsp2;Parent=match_sequence2;Target=EST:NM_000492+133+4575
ref|NM_001007143.1|     BLASTN  match   1       4332    5019    +       .       ID=match_sequence3;Target=EST:NM_000492+133+4455
ref|NM_001007143.1|     BLASTN  HSP     1       4332    2532    +       .       ID=match_hsp3;Parent=match_sequence3;Target=EST:NM_000492+133+4455
```

```
ref|NM_000492.2|        BLASTN  match   1       6129    1.201e+04       +       .       ID=match_sequ
ref|NM_000492.2|        BLASTN  HSP     1       6129    6060    +       .       ID=match_hsp1;Parent=
ref|NM_001032938.1|     BLASTN  match   1       4446    8187    +       .       ID=match_sequence2;Tc
ref|NM_001032938.1|     BLASTN  HSP     1       4446    4130    +       .       ID=match_hsp2;Parent=
ref|NM_001007143.1|     BLASTN  match   1       4332    5019    +       .       ID=match_sequence3;Tc
ref|NM_001007143.1|     BLASTN  HSP     1       4332    2532    +       .       ID=match_hsp3;Parent=
ref|NM_174018.2|        BLASTN  match   54      5760    3253    +       .       ID=match_sequence4;Tc
ref|NM_174018.2|        BLASTN  HSP     54      2705    1641    +       .       ID=match_hsp4;Parent=
```

How to approach perl pipelines

- use strict and warnings
- use (bio)perl as glue
- http://www.bioperl.org/wiki/Main_Page
- google.com
- test small pieces as you write them (debugger: `perl -d`)
- construct a command line and test it (catch failure `...or die...`)
- convert into system call, check it worked with small sample dataset
- extend to more complex code only as needed
- if you use code more than once, put it into a subroutine in a module e.g. Util.pm
- get command line arguments with GetOptions()