Perl 1 – Introduction to Perl and programming
variables, loops, and input/output
BIOC 8142 Feb 18, 2013
Bill Pearson wrp@virginia.edu

- A quick introduction to Perl
  - Running perl
  - Variable types: `$scalars, @arrays, %hashes`
  - Flow control: `if/then/else, for, while`
  - Input/output: <> and `print; open()`
  - Useful perl functions: `split(), system()`
- Programming – the problem solving approach

1

What should you do to reinforce the lecture material?

- Beginner's introduction to Perl
  www.perl.com/pub/2000/10/begperl1.html
- Curtis Poe's Beginning Perl: (ch. 3 – 6)
  proquest.safaribooksonline.com/book/programming/perl/9781118235638
- Chromatic's Modern Perl (free for download or read online):
  http://onyxneon.com/books/modern_perl/
- Learning Perl (4th Ed) Schwartz, Safari Technical Books
  proquest.safaribooksonline.com/book/programming/perl/0596101058
- Programming Perl (4th Ed) Wall et al, Safari Technical Books
  proquest.safaribooksonline.com/book/programming/perl/0596000278
  Chapter 1, Introduction to Perl
- `man perlintro`
- `man perldebug`
- A nice book about programming (but in python)
  `www.openbookproject.net/thinkcs/python/english2e/index.html`

2

Running Perl

Running a script:

```
% perl myscript.pl
```

Perl "one-liners":

```
% perl -e 'print "Howdy\n";'
```

Spontaneous Perl:

```
% perl
Print "Here we are.\n";
<ctrl-D>
```

Executable scripts:

```
% chmod +x myscript.pl
% myscript.pl
```

Getting Help

Top-level help:

```
% perldoc perl
```

General help:

```
% perldoc perlfunc
```

Help with a specific function

```
% perldoc -f print
```

Search FAQ by keyword:

```
% perldoc -q sort
```

A Perl Template

```
#!/usr/bin/perl -w
use strict;       # required
use diagnostics; # optional

# your program begins here:
print "Goodbye, cruel world.\n";
```

Minimal perl

- Variables:
  - `$simple; @array ($array[$element]); %hash ($hash{$key})`
- Loops:
  - `while (condition) {};  for (my $i=0; $i<$n; $i++) {}`
  - `next; break;`
- Conditionals:
  - `if (condition1) {...} elsif (condition2) {} else {}`
  - `if ($line =~ m/^>/) {next;}`
- Input/Output:
  - `while (my $line = <>)`
    `{ chomp($line); my @data = split(/\t/,$line);}`
  - `open($fd, "my_data.dat")||die "Cannot open my_data.dat";`
    `while (my $line = <$fd>) {...}`
  - `print join("\t",@data),"\n";`
- Regular expressions:
  - `m/^(\w+)(\d+)\s/; # $1 has (\w+)  $2 has (\d+)`
  - `s/"//g   # substitute: delete "'s`

Variables – scalars, lists, and hashes

- All programming languages (and programmable calculators) have places to store information–numbers or strings
- In 'C' or 'java', variables must be pre-declared, and look like any other "words" in the language: `int i, j; float mean;`
- Perl variables can be pre-declared (they must be pre-declared with `"use strict"`) but some programs do not pre-declare them
- Perl variables *must* begin with `"$"` (scalar), `"@"` (array), or `"%"` (hash). This symbol tells you how the variable will be used, but not necessarily what it is. Thus, $a[3] is a scalar from an array; $h{key} is a scalar from a hash.
- Scalar variables begin with `'$'` – `$i, $j, $mean`
- a scalar variable can hold an integer `$i=1;` real `$pi=3.14159;` or string `$sequence="acgttcggaccctgat";`
- a computer program variable is a place to store something, not an arbitrary value (constant) like in mathematics.

Literals: strings and numbers

```
% perl –e 'print 2 + 2; print "\n";'
% perl –e 'print "abc"; print "def\n";'

# string “addition” (concatenation operator)
% perl –e 'print "one two" . " and three\n";'

# mixing numbers and strings:
% perl –e 'print (2 * 2) . “\n”;’
% perl –e 'print "2 + 2 = " . (2 + 2) . "\n";'

# decimals and concatenations:
% perl –e 'print 2.3 + 2 . "\n";'
% perl –e 'print 2 . 3 + 2."\n";'
```

Arrays (lists, vectors)

- Most computer programs manipulate more complex (or structured) data – lists or vectors, matrices (2D arrays), etc.
- Perl provides a 1-d array or vector (list) data type; more complex 2-d structures built from 1-d arrays of references to other arrays.
- Perl list variables (arrays) begin with `'@'` :
  `@nt=('a','c','g','t')`
- lists (arrays) have individual elements, and a size (or length)
- An array (list) element is always a scalar, and is referred to as: `$nt[0]`, `$nt[1]` , etc. Scalars always begin with `'$'. '@'` things are always lists (never scalar array elements).
- Arrays (lists) begin with element `0 $list[0]`
- Perl has the confusing notion of "context", so that the statement

  ```
  while($i < @array) {...}
  ```

  makes sense. In this case, `@array` is evaluated in "scalar context", which provides the number of elements in `@array`.

  ```
  while($i < scalar(@array)) {...}  #easier to remember
  ```

Initializing arrays and manipulating lists (arrays)

```
@list=(1,2,3,4,5);     # all elements of the same type (int)
@list=(1,3.14159,"Pi");  # three different types
@nt=('a','c','g','t'); # DNA
@nt=qw( a c g t );            # qw means no ' or " or ','
@purine=qw(a g); @pyrimidine=qw(c t);
@nt = (@purine, @pyrimidine)==
          ('a','g','c','t');  # Lists combine or "flatten"
$a = 'a'; $c='c'...; @nt=( $a, $c, $g, $t );  # interpolation
($a,$c,$g,$t) = @nt;  # assigning to lists
@lines = split("\n",$lots_of_lines);
@words = split(" ",$lots_of_words);
@letters = split("",$string); # nothing in ""
@nt = split("","acgt");  but usually it's split(//,"acgt");
```

Perl operators (`man perlop`)

- Arithmetic: addition, subtraction, multiplication, division, modulus (remainder)

```
$a = 2 + 2;
$a = 2 + 2 * 2; $a = 2 + (2 * 2);
# operator precedence, use parens
$i++, $i-- # autoincrement by one (++$i, --$i)
$c += ($a + $b) # increment by ($a+$b)
```

- Logical: and `'&&'` or `'||'` not `'!'`
- Comparison

`>, >=, ==, !=, <=, <` for numbers `$a=12345;`

`gt, ge, eq, ne, le, lt` for strings `$a='catdog';`

11

Flow control: `if/else`

- Standard (compound) `if/unless/elsif/else`

```
my ($sig, $border, $not_sig) = (0,0,0);
if ($evalue <= 0.001) { $sig++;}
elsif ($evalue <= 0.1) { $border++;}
else {$not_sig++;}

unless (condition) == if (!condition)
```

- Inverted (simple) `{ } if/unless`

```
$border++ if ($evalue <= 0.1);
$sig++ unless ($evalue > 0.001);
```

How do these values ($sig, $border) differ in the "Standard" and "Inverted" cases (will `$border` be the same? `$sig`?).

12

Flow control: for () {} loops

```
my $sum = 0; # always remember to initialize
for my $value ( @array_of_values)
{ $sum += $value;}

for (my $i=0; $i < scalar(@array); $i++)
{ $sum += $array[$i];}
```

in perl, indexing is mostly used if you need a previous or next value, or the index itself

13

Flow control: while () {} loops

```
while ( my $line = <>) {
  chomp($line) # always remove "\n" first
  @columns = split(/\t/,$line);
}
```

- 'next' skips the rest of the loop

```
while ( my $line = <>) {
  next if ($line =~ m/^>/);
# match '>' at beginning of line, FASTA header
  $cnt++;
}
```

- 'last' exits the loop

```
while ( my $line = <>) {
  chomp($line) # always remove "\n" first
  @columns = split(/\t/,$line);
  last if ($columns[-2] > 0.001);
}
```

14

Input/Output I

- Data is read with the the '<>' operator (<> or <INFILE>)

```
while (my $line = <>) { . . . }
```

  - If you put a file name on the command line, <> reads from the file
  - If you put a list of files <> reads them in order
  - If you don't put a file, <> uses stdin
- Lines read from files have "\n" at the end; always remove it

```
chomp($line);
```

- Every <> reads a line (use it alone to skip lines)
- If only one file read at a time, and one file for output, use <> for input and '>' on the command line to write to `stdout`.

Input/Output II

- The <> input operator can also be specific to a file name by included a "file handle"
- `open(INFILE, "<gstm1_human.aa");` – open file for reading
-
```
$file = $ARGV[0];
open (INFILE, "< $file")||
   die ("Cannot open $file");
```
- `$line = <INFILE>;` – read a line
- `chomp($line);` – remove "\n"
- `while ($line=<INFILE>) { . . . }`
- `close(FILE); # files are closed automatically when the program ends`

Input/Output III

- Files can also be opened for writing ('>' or '>>' – extend at end)
  - `open (OUTFILE, "> $file); # open file for writing`
- To send data to a file (or stdin), use 'print'
  - `print "accn: $prot_acc evalue: $evalue\n";`

  Goes to `stdout` (> on the command line)
  - `print OUTFILE "accn: $prot_acc . . .\n";`

  Goes to the file specified by `open(OUTFILE,">$file")`
- Output lines always need `"\n"`
- If input is <tab> delimited, output often is as well.

  `print join("\t",($query_acc, $subj_acc, $evalue)),"\n";`
- `close(FILE);`

Input/Output Examples

Count lines in a file:

```
$ perl -e '$c=0; while (<>) { $c++; } print "$c\n";' gstm1_human.aa
5
$ wc gstm1_human.aa
  5  14 311 gstm1_human.aa
$ perl -e '$c=0; while (<>) {$c++;} print "$c\n";' gstm1_human.bl_tab
146
$ wc gstm1_human.bl_tab
  146  1752 13975 gstm1_human.bl_tab
```

18

Input/Output Examples

Count words in a file:

```
$ perl -e '$c=0;
> while ($l=<>) {
> chomp $l;
> $c += scalar(split(/\s+/,$l));
> }
> print "$c\n";' gstm1_human.bl_tab
1752
$ wc gstm1_human.bl_tab
  146  1752 13975 gstm1_human.bl_tab
```

19

Summarize blast output with perl

Problem – write a Perl script to identify distant homologs, and re-search swissprot with those sequences

- Did problem before with bash

  `blastp –outfmt 6, cut –f 2`

- With perl, we can look at the expectation value to find distant homologs (0.001 < evalue < 2.0)

20

Perl problem solving – initial steps

1. Look at the (raw) data
2. Identify what we need
3. Isolate the numbers needed, and save them
4. Do the necessary arithmetic/selection

1) Look at the data

```
sp|GSTM1_HUMAN   sp|GSTM1_HUMAN 100.00 218  0   0  1  218 1 218 7e-127 452
sp|GSTM1_HUMAN   sp|GSTM4_HUMAN  86.70 218 29   0  1  218 1 218 3e-112 403
sp|GSTM1_HUMAN   sp|GSTM1_MACFA  85.78 218 31   0  1  218 1 218 3e-110 397
sp|GSTM1_HUMAN   sp|GSTM2_PONAB  85.78 218 31   0  1  218 1 218 1e-109 395
sp|GSTM1_HUMAN   sp|GSTM2_MACFA  85.78 218 31   0  1  218 1 218 1e-109 395
sp|GSTM1_HUMAN   sp|GSTM5_HUMAN  87.61 218 27   0  1  218 1 218 1e-109 395
```

```
blastp –help
*** Formatting options
-outfmt <String>
  alignment view options:
    0 = pairwise,

    5 = XML Blast output,
    6 = tabular,

  When not provided, the default value is:
  'qseqid sseqid pident length mismatch gapopen qstart qend sstart send
  evalue bitscore', which is equivalent to the keyword 'std'
  Default = `0'
```

2) Identify/extract the data we need

qseqid	sseqid	pident	len	mis	gp	qs	qe	ss	se	evalue	bits
sp|GSTM1_HUMAN	sp|GSTM1_HUMAN	100.00	218	0	0	1	218	1	218	7e-127	452
sp|GSTM1_HUMAN	sp|GSTM4_HUMAN	86.70	218	29	0	1	218	1	218	3e-112	403
sp|GSTM1_HUMAN	sp|GSTM1_MACFA	85.78	218	31	0	1	218	1	218	3e-110	397
sp|GSTM1_HUMAN	sp|GSTM2_PONAB	85.78	218	31	0	1	218	1	218	1e-109	395
sp|GSTM1_HUMAN	sp|GSTM2_MACFA	85.78	218	31	0	1	218	1	218	1e-109	395
sp|GSTM1_HUMAN	sp|GSTM5_HUMAN	87.61	218	27	0	1	218	1	218	1e-109	395

1. Subject accession (sseqid)
2. evalue
3. Select hits with 0.01 < evalue < 2.0

2) Identify/extract the data we need

qseqid	sseqid	pident	len	mis	gp	qs	qe	ss	se	evalue	bits
sp|GSTM1_HUMAN	sp|GSTM1_HUMAN	100.00	218	0	0	1	218	1	218	7e-127	452
sp|GSTM1_HUMAN	sp|GSTM4_HUMAN	86.70	218	29	0	1	218	1	218	3e-112	403
sp|GSTM1_HUMAN	sp|GSTM1_MACFA	85.78	218	31	0	1	218	1	218	3e-110	397
sp|GSTM1_HUMAN	sp|GSTM2_PONAB	85.78	218	31	0	1	218	1	218	1e-109	395
sp|GSTM1_HUMAN	sp|GSTM2_MACFA	85.78	218	31	0	1	218	1	218	1e-109	395
sp|GSTM1_HUMAN	sp|GSTM5_HUMAN	87.61	218	27	0	1	218	1	218	1e-109	395

Get the data:

```
while (my $line = <>) {
   chomp($line);
   my @data = split(/\t/,$line);
...
```

3) Isolate the numbers, and save them

```
qseqid          sseqid          pident len mis gp qs qe ss se evalue bits
sp|GSTM1_HUMAN  sp|GSTM1_HUMAN 100.00 218  0  0  1 218 1 218 7e-127 452
sp|GSTM1_HUMAN  sp|GSTM4_HUMAN  86.70 218 29  0  1 218 1 218 3e-112 403
```

How do we refer to the data? `@data = split(/\t/,$line);`

1) Array:

```
   $data[0], $data[1], $data[3], ...
```

or isolate the ones you need: (array slice, just pick what you want)

```
   my @hit_data = @data[1,10];
      @hit_data = @data[1,-2];
```

The problem with arrays is that you need to remember where the data is. Is $data[10] the evalue, or the bit score?

2) Hash:

```
my %hit_hash = ();  # read about "hash slice"
@hit_hash{qw(qseqid sseqid ... evalue bits)} = @data;
   # or just use split(/\t/,$line)
```

4) Do the necessary arithmetic/selection

```
qseqid          sseqid          pident len mis gp qs qe ss se evalue bits
sp|GSTM1_HUMAN  sp|GSTM1_HUMAN 100.00 218  0  0  1 218 1 218 7e-127 452
sp|GSTM1_HUMAN  sp|GSTM4_HUMAN  86.70 218 29  0  1 218 1 218 3e-112 403
```

Identify the hits with 0.01 < evalue < 2.0

```
if ($hit_hash{evalue} > 0.01 &&
    $hit_hash{evalue} <= 2.0) {
  print "$hit_h{sseqid}\t$hit_h{evalue}\n";
}
```

4) Do the necessary arithmetic/selection

Col/field:0	1	2	3	4	5	6	7	8	9
"ProbeName"	"GeneN"	"FeatNum"	"SysName"	"logFC"	"AveExpr"	"t"	"P.Value"	"adj.P.Val"	"B"
"A_23_P53476"	"LDHB"	26180	"NM_002300"	-6.42250	10.19696	-45.976	1.188e-08	7.131e-05	10.483
"A_24_P914434"	"GSTM3"	6712	"NM_000849"	-6.25976	10.67475	-45.273	1.299e-08	7.131e-05	10.422
"A_23_P134426"	"GPNMB"	26129	"NM_001005340"	-7.11150	10.19522	-43.103	1.725e-08	7.131e-05	10.225
"A_24_P625382"	"CSDA"	34396	"NM_003651"	-5.41279	9.980151	-40.891	2.339e-08	7.131e-05	10.004

Find the gene with the most probes:

```
my %probe_count = ();

if ($probe_count{$data_columns[1]}) {
    $probe_count{$data_columns[1]}++;
}
else {
    $probe_count{$data_columns[1]} = 1;
}
```

Homework I

1. Write a perl program to print out 20 random numbers.
   - Modify the program to print out numbers between 0 and 10
   - Modify the program to print out "whole" numbers between 0 and 10
   - Check that it is working by printing out an extra line that reports the maximum of the 20 numbers, and the minimum.
2. Modify the program to calculate the average of the 20 numbers.
   - What is the average when you average all numbers generated between 0 and 10.
   - What is the average when you calculate only "whole" numbers?
3. Modify the program to calculate the median of 20 random numbers between 0 and 10

   (hint, use an @array, and sort the @array numerically; perldoc –f sort; be sure the print out the sorted array to make sure it worked.)

28

Homework II

Reproduce your bash scripts with a Perl script:

1. Write a perl script to extract from the blastp tabular hits the "sseqid" and "evalue" for hits with 0.01 < evalue <= 2.0
2. Modify the script to extract the GI and the accession (P12345, without the version number), from the "sseqid"
3. Use the 'system()' function (perldoc –f system) to use 'curl' to download the sequences from the NCBI and save them in a file named after their accession number
4. Again, using the "system()" function, run the blast searches for the sseqid's with 0.01 < evalue<=2.0 against SwissProt, saving the results in a file named after the original sseqid accession (P12345) number and the suffix ".bp"

29