Perl Programming Fundamentals for the Computational Biologist

Class 2

Marine Biological Laboratory, Woods Hole
“Advances in Genome Technology and Bioinformatics”
Fall 2004

Andrew Tolonen
Chisholm lab, MIT

Review from class 1

Perl has three types of variables: $scalars, @arrays, and %hashes.

Control structures (if...else, while, for, foreach, etc.) allow you to impose logic on your scripts.

Filehandles are how you connect your perl script to the outside world (the keyboard, the screen, a file on your computer).

```
open (IN, “<./inputfile.txt) or die “can't open the input file\n”;
```

You can search for a pattern in a string by doing a pattern match.

```
if ($string =~ /pattern/)
{
  # do something
}
```

Regular Expressions

You use regular expressions to specify a pattern to be matched against a string.

Regular expressions (Regexs) are one of Perl's strongest features. They are also some of the most confusing. Regexs are a language in themselves!

The =~ operator: use the binding operator to match a regex against a string

Regexs are great because they use symbols that allow you to specify patterns of arbitrary complexity.

Regular expression symbols

Table of important regex symbols

Symbol	**Meaning**
.	any single character
*	zero or more of previous character

+	one or more of previous character
?	zero or one of previous character
{MIN, MAX}	match between MIN and MAX times
[ ]	character class
\d	digit (same as [0-9])
\s	whitespace
\S	not whitespace
\t	tab
\w	any word character
^	match at the start of string
$	match at the end of string

Examples

```
$string =~ /taat./;
         # match taat followed by any single
                  # character
$string =~ /[ta]ttaat/;
         # match “tttaat” or “attaat”
$string =~ /\s*Gene/;
         # match zero or more whitespaces followed
            # by “Gene”
$string =~ /\s+Gene/;
         # match one or more whitespace followed by
              # ”Gene”
$string =~ /t{3,5}aatat/;
        # match anywhere between 3 and 5 t's
               # followed by  “aatat”
$string =~ /^>Gene 1/;
        # match strings that start with the pattern
         # “>Gene 1”  i.e in FASTA files
$string =~ /annotation$/;
        # match strings that end with the word
                  # annotation
```

Backreferences to regexs

- Putting a regex in () saves the matched string for later retrieval, called a back reference.

- Backreferences are retrieved as the variables $1 for the first saved match, $2 for the second saved match,
etc.

Example

```
#!/usr/bin/perl

# this is a script to read a sequence file and #report each
line that contains a given binding  #site motif.  It uses
backreferences so that the
# actual motif found is printed.

open (SEQ, "<./sequence.fasta") or die "cant open
sequence file\n";

$bindingsite = "tgta[acgt]{5,8}ta[cg]a";
$linecounter = 1;  # keeps track what line we're on

while ($line = <SEQ>)
{
  if ($line =~ /($bindingsite)/)
  {
   print "line number $linenumber contains the motif $1;
  }
$linecounter++;
}
##############################################
```

Subroutines

- Functions such as **chomp()** are system-defined functions. Subroutines let you define your own functions.
- Subroutines make it more efficient to reuse batches of code in your programs.
- Subroutines allow you to treat aspects of your programs like a black-box (data abstraction).

Here is an example:
Lets's say you are writing a script where you repeatedly want to find the sum of three numbers. Instead of rewritting how to calculate the sum, write a subroutine!

```
 #!/usr/bin/perl

# this is a script to find the sum of three numbers

$num1 = 1;
$num2 = 2;
$num3 = 3;

$mysum = sum($num1, $num2, $num3);
print "the sum of $num1, $num2, and $num3 is $mysum\n";


sub sum
{
 $thesum = $num1 + $num2 + $num3;
 return $thesum;
}
```

Subroutines are called within any expression by following the subroutine name with (). Some folks also recommend to precede the name with &. This avoids conflicts when there is a system-defined function with the same name.

```
SubroutineName();
```
or
```
&SubroutineName();
```

You often want a subroutine to return a value. The return value is the last expression evaluated in the subroutine.

Passing arguments to subroutines

You can pass arguments to subroutines by putting them in parentheses after the subroutine call. Any arguments passed to the the subroutine come in as the @_ array.

Example

```
#!/usr/bin/perl

# this script uses the findMax() subroutine to find # the
max of a set of user defined numbers.

# prompt the user for a list of numbers
```

```
print "Please enter a list of numbers, each separated by a
space: ";
$num = <STDIN>;

# die if the list contains entries other than
# numbers
unless ($num =~ /^[\d\s]+$/)
{
  print "that is not a valid list!\n";
  exit;
}

# split the numbers into an array
@nums = split(" ", $num);


$max = findMax(@nums);
print "The max number in your list is $max\n";

sub findMax
{
# this is a subroutine to return the max of a list # of
numbers
 @nums = @_;  # pass the array into the subroutine
 $test = $nums[0];
 foreach $elt (@nums)
 {
  if ($elt > $test)
  {
   $test = $elt;
  }
 }
 return $test;
}
```

Private variables in subroutines

It is often useful to make variables private to your subroutine. This means that they can't be changed by commands outside the subroutine. Use **my** to confine a variable to a block of code {}.

Example:

```
sub findSum
# this is a subroutine to return the sum of a list
```

```
# of numbers. The subroutine takes an array of
# numbers as input and returns the sum.
{
  my ($elt, $sum);
  my (@nums);
  @nums = @_;
  foreach $elt (@nums)
  {
    $sum = $sum + $elt;
  }
  return $sum;
}
```

Passing by reference

If you want to pass more than one array or hash to a subroutine, you need to pass them by reference. This is because the arguments all get squished into @_ when they are passed to the subroutine. By using references to pass pointers to the arrays/hashes you can preserve them.

Example

```
#!/usr/bin/perl

# this script illustrates how to pass multiple
# arrays to a subroutine
# by passing them by reference

# make 2 simple lists of gene names
@list1 = ("Gene A", "Gene B", "Gene C");
@list2 = ("Gene D", "Gene E", "Gene F");

getLast(\@list1, \@list2);

sub getLast
{
 # this subroutine prints the last element in each
 # list
 my ($ref1, $ref2, $last1, $last2);
 my (@list1, @list2);

 ($ref1, $ref2) = @_;
 @list1 = @$ref1;
 @list2 = @$ref2;
 $last1 = pop(@list1);
```

```
 $last2 = pop(@list2);
 print "the last elt in the first list is $last1\n";
 print "the last elt in the second list is $last2\n";
}

#####################################################
```

Subroutines in external files (libraries and modules)

You can place multiple subroutines in separate files and access them in your script. By convention, you should use the file extensions .pl (libraries of subroutines) or .pm (perl modules)

Perl libraries

The **require** keyword loads a file into your program

Example:

Here is the executable script findMax.plx

```
#!/usr/bin/perl

require "./findMax.pl"; # script requires a perl library

@nums = (1, 2, 3, 12, 35, 4);

$max = findMax(@nums);

print "the max number in your list is $max\n";
```

Here is the library file findMax.pl in the same directory

```
Sub findMax
{
# this is a subroutine to return the max of a list of
numbers
 @nums = @_;
 $test = $nums[0];
 foreach $elt (@nums)
 {
  if ($elt > $test)
  {
   $test = $elt;
```

```
  }
 }
 return $test;
}
# Note!  the last line in the file is "1"
1
```

Perl modules

- Perl modules are batches of reusable code.
- You can download thousands of modules from CPAN (Comphrehensive Perl Archive Network) www.cpan.org

Installing modules on a linux machine

- download the module from cpan.org
- unzip the module
  ```
  % gunzip moduleName.tar.gz
  ```
- extract the module
  ```
  % tar -xvf moduleName.tar
  ```
- Build the source code
  ```
  % perl Makefile.pl
  % make
  % make test
  % make install
  ```

Using Perl Modules

- modules are included in your script by using the **use** command

Example: use LWP to fetch the contents of a web page from within a Perl script.

```
#!/usr/bin/perl

use LWP::Simple;

# this is a script to grab a webpage and print it
# to a local file

$url = "http://www.cnn.com";
$page = get($url);

open (OUT, ">./webpage.html") or die "cant open out\n";
```

```
print OUT “$page;

close OUT;
```