Introduction to Perl

Why is perl useful for biologists (including bioinformatics people)? You can do a lot of bioinformatics research without programming since there are a lot of existing softwares. But what if you can't find a program which will do exactly what you want? Or you want to do some simple analysis, which can be done with pre-existing, but a complicated program. It is sometime quicker to make the simple analysis program than trying to spend hours learning how to use the pre-existing program, whose features you probably don't need in the future.

More frequently, I encounter situations where there are several steps for the analysis, and the steps are done by different pre-existing programs. Those patient people can sit in front of the computers for hours, and repeat the steps by hand. If you find a mistake in your analysis, you can start over from the beginning by hand. But with perl, you can join the programs and automate the analysis: i.e., create a small program, which will drive the other programs. The analysis is repeatable, and easy to fix problems, or redo similar analysis with many data sets.

Perl is a wondeful, full-blown programming language, but I frequently use it to make SMALL programs for this kind of quick automation. You can be lazy now and not to learn perl programing, or you can be lazy later after you learn perl. Who saves more time over a long run?

- Perl is a ”Practical Extraction and Report Language”
  - Practical (easy to use).
    You can write small programs very quickly. Low learning curve.
  - Originally developped for text manipulations, (e.g., extract the 2nd column from a tab-delimited text file, and print it out in comma separated fields).

    But it is a general, complete programming language.
- Wide user base, so easy to get help. Also many extensions are available (e.g. bioperl ¡http://www.bioperl.org¿).
- Extensively used for programming World Wide Web electronic forms
- Easy to “glue” several programs and databases.

  e.g., automatically download sequences from genbank, run multiple sequence alignment with clustal, and reformat it to nexus file, and run it with paup. You can make a perl program, which will do all of these in one command.
- Open source, and free. Runs on most platforms.

Goal: Learn how to modify a script to:

1. Extract information from a text output of other programs
2. Create a simple program to run simulatios for approximate bayesian computation (ABC).

1 First encouter

- Type in the following perl script (e.g., `emacs hw.pl`).

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

print ("5 beer, please.\n");
```

```
$drink = "whisky";
$howmany = 1;
$howmany = $howmany + 1;

print ("Or $howmany $drink, please.\n");

exit;
```

- Now, from the shell (command line terminal), turn on the executable bits.

  ```
  $ chmod +x hw.pl
  $ ls -l
  -rwxr-xr-x  1 ntakebay ntakebay 45 Apr  9 20:00 hw.pl
  ```

  Remember the unix **permissions**? To run a script, the file has to be **executable** (x). If you use minus sign (`chmod -x hw.pl`), the executable bit get unset.

`chmod u+w file`	(u)ser who owns it can write
`chmod o+r file`	(o)ther people can read
`chmod ugo-wx file`	(u)ser, (g)roup, and (o)thers cannot write or execute
`chmod -wx file`	same as above

- To run it, type `./hw.pl`
- Details:
  - The first line starting from `#!` tells that the script should be "interpreted" by a "perl interpreter" program `/usr/bin/perl`.
    If the perl interpreter program `perl` is not at this location, it might be installed in `/usr/local/bin/perl`. Then the first line should be "`#!/usr/local/bin/perl`".
    To find the location of the perl interpreter, try

    ```
    which perl
    ```
  - `-w` tells that the perl interpreter will print out extra warnings (easier to catch bugs).
  - Each **statement** is terminated by a **semi-colon** (;).
  - Text string is surrounded by double-quotes (").
  - A simple variable (= a container/memory to store data) starts with $.
  - I used `howmany` as the name of the container (variable). But you can give whatever name to each container (e.g. "`$varA`", "`$weight1`").
  - You can change the contents of variables.

2 Example: Extracting information from a text file

A practice to get some information out of text file.

- This commands:

  ```
  ms 30 4 -t 3.0 | sample_stats
  ```

  prints out:

  ```
  pi: 4.232 ss: 13 D: 0.956 thetaH: 3.285 H: 0.947
  pi: 1.314 ss: 5 D: 0.114 thetaH: 0.478 H: 0.836
  pi: 2.540 ss: 8 D: 0.783 thetaH: 1.942 H: 0.597
  pi: 2.726 ss: 14 D: -0.762 thetaH: 1.549 H: 1.177
  ```

  We are making a script to extract desired columns (only the values).

- Simple perl script (cleanSampleStatsSimple.pl):

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

  while(<>) {            # read in each line
      if (/pi:\s+(\S+)\s+ss:\s+(\S+)/) { # Regular Expression
  my $pi = $1;
  my $ss = $2;

  print join("\t", ($pi, $ss));
  print "\n";
      } else {
  warn "WARN: This line does not follow the normal output pattern, " .
      "Ignored...:\n$_";
      }
  }

  exit;
  ```

- This can be used in the following way:

  ```
  ms 30 4 -t 3.0 | sample_stats | ./cleanSampleStatsSimple.pl
  ```

- You can get the same results with:

  ```
  ms 30 4 -t 3.0 | sample_stats | cut -f 2,4
  ```

2.1 Details: line-by-line processing, conditional, regular expressions

- Diamond operator

  ```
  while(<>) {
   ....;
   ....;
  }
  ```

  - This is used to process each line by line.

  - A special variable `$_` contains a single line.
  - The process will keep looping until all lines are read.

  Example:

```
print "Starting\n";
$counter = 1;
while (<>) {
  print "$counter : $_";
  $counter = $counter + 1;
}
print "End of file\n";
```

- Conditional:

```
if (condition){
  do this 1;
  do this 2;
} else {
  do this 3;
  do this 4;
}
```

  If the *condition* (e.g. `$num < 0`) is true, it will do 1 & 2, and skip 3 & 4. If the *condition* is false, it will do 3 & 4 only.

  Example:

```
$num = - 0.4
if ($num < 0) {
  print "$num is negative\n";
} else {
  print "$num is NOT negative\n";
}
```

- Pattern matching with Regular Expressions

```
if (/ ..... /) {
  do this 1;
} else  {
  do this 2;
}
```

  - For the *condition*, you can use some pattern enclosed by `/ /`.
  - It checks if the special variable `$_` contains the specified pattern.
  - Examples;
    * Text string
      `/dog/`
      This matches if `$_ = "hotdog eating champ"`. Or any text containing `dog`.

∗ Multiplier

  `/Hmm+/`

  Plus (+) means one or more of the *immediately previous* character.
  It mathces "Hmm" or "Hmmm" or "Hmmmmmmm".
  But not "Hm"

∗ Special character

```
/\s/   # A "space" character, space, tab, newline
/\S/   # A non-space character (A, a, 1, 0, :, _, etc
/\d/   # A digit (0, 1, ... 9)
/./    # matches any single character
```

∗ Extraction

  Parentheses in a regular expression is used to extract information

```
# extract hours, minutes, seconds
if ($time =~ /(\d\d):(\d\d):(\d\d)/) {  # match hh:mm:ss format
  $hours = $1;
  $minutes = $2;
  $seconds = $3;
}
```

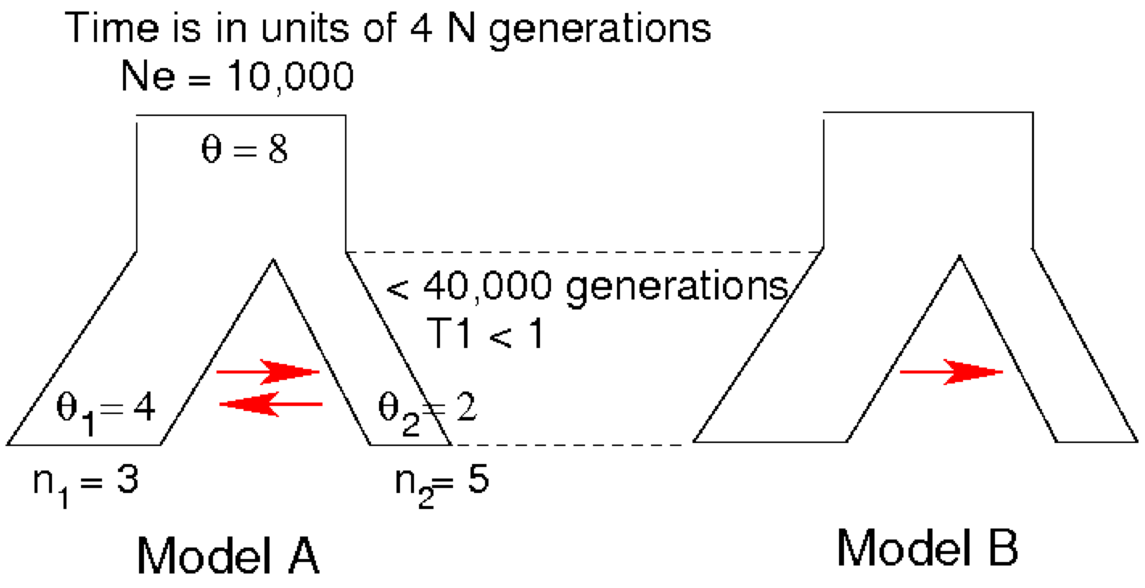
- More information about regular expressions.
  - Official tutorial: ¡http://perldoc.perl.org/perlretut.html¿
  - Shorter tutorial: ¡http://www.faculty.uaf.edu/ffnt/teaching/programming/perl2nd/index.html¿

2.2 Exercise:

Modify the script to extract all five summary statistics.

3 Example: Drive other programs

- Simple divergence model.



- Goal: Test if symmetric migration model or unidirectional model is better.
- Strategy:

  1. Choose a random number (0.01 - 1) for T1
  2. Run 1 `ms` after flipping a coin:
     head -¿ Simulate Model A

     ```
     ms 8 1 -t 8 -I 2 3 5 -ma x 8.0 16.0 x  -n 1 0.5 -n 2 0.25  -en $T1 1 1.0  -ej $T1 2 1
     ```

     tail -¿ Simulate Model B
     Use the migration matrix:

     ```
     -ma x 8.0 0.0 x
     ```

  3. Calculate the summary statistics (pi, number of segregating sites, D, thetaH, H).
  4. print out following info:

     – model (0 for model-A, 1 for model-B)

     – T1 (drawn from the prior distribution)

     – a vector of summary statistics
  5. Goto Step 1 and repeat this many many times.

  The final output will look like:

```
M T1 pi ss D thetaH H
0 0.605 2.07 6 -0.48 1.35 0.71
0 0.034 18.75 40 1.15 17.53 1.21
1 0.045 8.32 24 -0.53 5.67 2.64
: :
: :
```

- Simulation code
- Migration matrix
  - `-ma x 8.0 16.0 x` is equivalent to:

$$\begin{pmatrix} 4N_0m_{11} & 4N_0m_{12} \\ 4N_0m_{21} & 4N_0m_{22} \end{pmatrix} = \begin{pmatrix} x & 8.0 \\ 16.0 & x \end{pmatrix}$$

  m_{ij} is the fraction of subpopulation *i* which was in the *j*-th subpopulation in the previous generation.

  - A little tricky because two different subpop sizes.
  - We specified $\theta = 4N_0\mu = 8$ (`-s 8`). This means that the ancestral population size = 1, N_1 = 1/2, and N_2 = 1/4.
  - So actual numbers of migrants into the subpop 1 & 2 are $2N_1m_{12} = 8.0/2/2 = 2.0$ and $2N_2m_{21} = 16.0/4/2 = 2.0$
  - So this is a symmetric migration model in terms of actual number of migrants.
  - more details in p. 12 of `ms` manual.

3.1 Background Info/Explanation of `msDrive.pl`

- For loop:

```
#!/usr/bin/perl -w
for ($i = 0; $i < 10; $i++) {
  $num = rand(5);
  print "repetion $i: $num\n";
}
```

In this loop, `$i` goes from 0 to 9.

3.2 Exercise: modify `msDrive.pl`

Instead of setting the migration rate to a pre-fixed value, modify the program to estimate the **migration rate** (in addition to divergence time). In other words, you need to draw the migration from a random distribution.