Introduction to Perl

HMS Research Computing

Fall 2017

Kathleen Keating

kathleen_keating@hms.harvard.edu

Overview

- Learn how to write and run Perl scripts on O2
- Become familiar with Perl syntax
- Understand the different data types in Perl
- Learn how to import and output files
- Example: parsing BLAST tabular output

Why Perl?

- Open source, and still under development
- Available for most operating systems: Windows, Mac, Unix
- Easy to learn, and scripts are quick to write
- Kitchen sink language
- TMTOWTDI – there's more than one way to do it
- Many bioinformatics modules and scripts available

What can you do with Perl?

- Data munging
  - Making your data set less “messy”
  - includes: combining, analyzing, filtering, reformatting data
- Automate analyses
  - Download many files from biological databases
- Interface with SQL databases
- Use other people’s code

Notation

- `[user123@login ~]$`

  O2 bash (your terminal)

- **1** `#!/usr/bin/env perl`

  2 `print "Hello world!\n";`

Perl script (you'd save this without line numbers and run)

- `my $scalar = 10;`

  Example Perl code snippet

Connecting to O2

Mac and Linux: Open a terminal, and ssh to O2:

```
$ ssh user123@o2.hms.harvard.edu
```

Windows: Use Putty to connect to O2.

Put `o2.hms.harvard.edu` as the Host Name (or IP address) and Select SSH as Connection Type

Once on O2, open an interactive shell:

```
[user123@login ~]$ srun --pty -p interactive -t 0-2
-n 1 bash
```

Where is Perl on O2?

- See available Perl modules

```
[user123@compute ~]$ module spider perl
```

- Load Perl module

```
[user123@compute ~]$ module load gcc/6.2.0 perl/5.
24.0
```

- Check which Perl will be used:

```
[user123@compute ~]$ which perl
```

```
[user123@compute ~]$ perl -v
```

General workflow

- Write Perl script in text editor (e.g. nano, vim, emacs)
  - Script should conventionally end in `.pl`
- Save your script, exit text editor
  - **Nano -** `CTRL + O` to save, `CTRL + X` to quit
- Run script on command line

```
[user123@compute ~]$ perl myscript.pl
```

Syntax

- Comments start with #
  - Can be a full line, or at the end of a set of commands
- Commands end in ;
- Use = to assign variables
- Whitespace doesn't matter to Perl, but should be used for clarity
- For printing – double quotes ("") interpolate variables, ('') do not
- Characters may need to be "escaped" for printing or regular expressions

First script

```
1 # This is a comment!
2 print “Hello world!\n”;
```

Open up a text editor and type the lines above.

(Do not add the line numbers in your script!)

Save as `hello.pl`. Run by:

```
[user123@compute ~]$ perl hello.pl
```

What is happening in the script?

```
1 # This is a comment!
2 print "Hello world!\n";
```

Comments start with `#`. Perl ignores these.

Each command needs to end in `;`

The `print` function sends text to the terminal, and `\n` is the newline character.

Functions take zero or more arguments

Shebang line

- The first line of your script describes what type of program it is, and how to execute it. This is called an interpreter directive.

```
#!/usr/bin/perl
```

- Or to make your program portable between Perl versions use:

```
#!/usr/bin/env perl
```

Variables

Scalars, Arrays, Hashes

Variable type: Scalars

- Hold one thing, such as a number, or text

```
$sequence = "GTCAGATTC";
$e_value = 1e-10;
```

- Name starts with $
- Can change the value of a scalar in a script

```
$gene_symbol = "BRCA2";
$gene_symbol = "INS";
```

Good practices:

- Declare variables with `my` the first time you use them, and don't need to give a value when it's declared.

```
my $sequence = "GTCAGATTC";
```

- Prevent misspellings:

```
use strict;
```

- Have Perl tell you when you're doing something wrong:

```
use warnings;
```

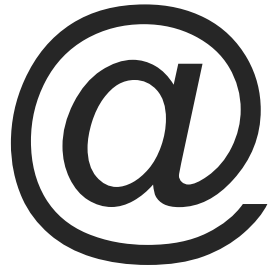
- `use` statements should be put at the top of your scripts.

Variable type: Arrays

- Hold a set of things, called elements

```
my @numbers = (1, 2, 3, 4);
my @scalars = ($a, $b, $c);
my @strings = (“abc”, “def”, “g”);
my @colors = qw(blue red green black);
```

- Name starts with @
- Common uses:
  - Saving contents of an input file
  - Performing an operation on all elements



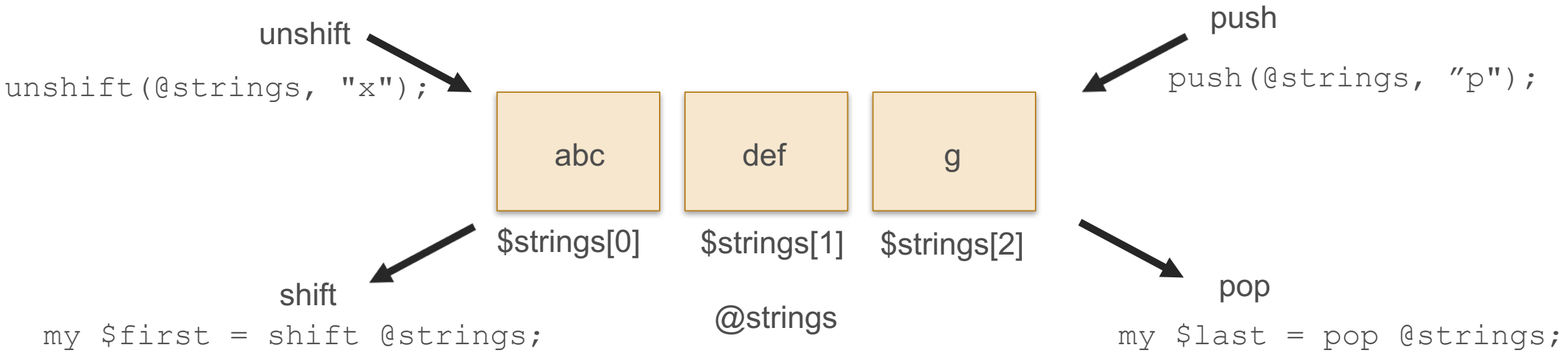
Arrays – what do they contain?

- Arrays have indexed elements
  - Each element is a scalar variable, so use $ not @
  - Indexes describe the element's position in the array, start from 0

```
my @strings;
@strings = (“abc”, “def”, “g”);
print $strings[2];
print @strings;
print “$strings[0]\t$strings[1]\t$strings[2]\n”;
print “Array size: ”, scalar(@strings),”\n”;
```

Array manipulation

- Arrays can change in size:



- Can also change a single value:

```
$strings[2] = "q";
```

Strings to Arrays

- Use the `split()` function to convert a string to an array
  - Need to split on a delimiter, such as tabs or spaces
  - Split a sequence on Cs:

```
my $sequence = "ATGCACGAA";
my @seq = split /C/, $sequence;
#("ATG","A","GAA")
```

  - Split a line on tabs:

```
my @columns = split /\t/, $line;
```

Special array @ARGV

- So far we have been using hard-coded variables. How do we accept user input? Use `@ARGV`!
- Anything you provide after the script name goes into `@ARGV`

```
[user123@compute ~]$ perl argv.pl "Abe Lincoln" 1809 blue

my $name = $ARGV[0]; # "Abe Lincoln"

my $year = $ARGV[1]; # 1809

my $color = $ARGV[2]; # blue
```

- Can alternatively use module `Getopt::Long` for command line options

Getopt::Long for command line parameters

- Getopt::Long is a module (library of functionality that someone else wrote) that allows you to input options on the command line

```
use Getopt::Long; # use the module
use strict;
my $bedfile = “file.bed”; #declaring default value
GetOptions('bed=s' => \$bedfile);
```

- =s is for text string, can also use =i (integer), or =f (float)
- now can specify a file with the –bed option on command line
  - Value will be assigned to $bedfile

Variable type: Hashes

%

- Hold a set of unordered key/value pairs
  - One value for key (unless using complex data structure)
  - Keys are unique – will overwrite if you give multiple pairs with the same key
- Name starts with %
  - When accessing part of hash $ is used
- General uses:
  - Holding varied data
  - Quickly searching through a dataset
  - Making a unique set

Declaring Hashes

- Method 1 – without assignment operator

```
my %sci_names = (“human”, “Homo sapiens”, “mouse”, “Mus
musculus”, “rat”, “Rattus norvegicus”);
```

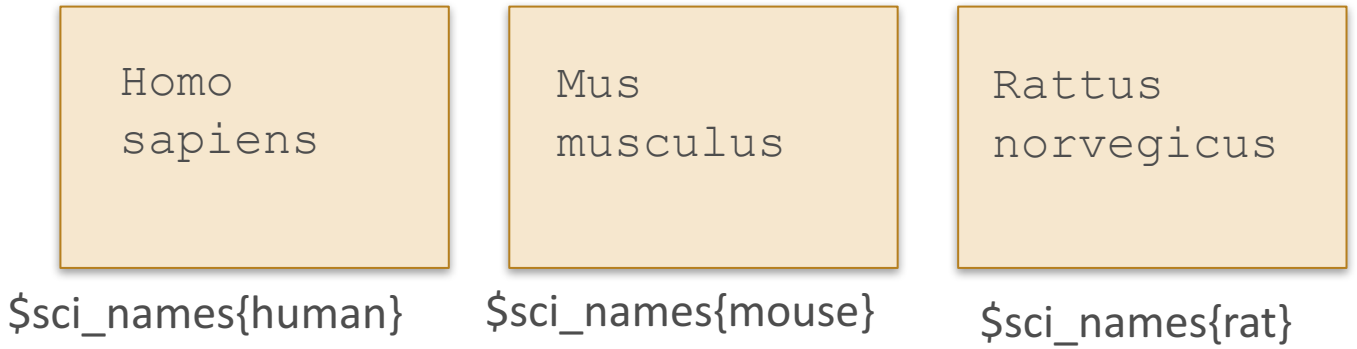
- Method 2 – with => operator

```
my %translate = (
   “ATG” => “M”, “GGT” => “G”,
   “CAT” => “H”, ”TAG” => “*”,
   …
)
```

- Adding one value: `$hash{$key} = $value;`

Accessing hash contents

- `my %sci_names = (“human”, “Homo sapiens”, “mouse”, “Mus musculus”, “rat”, “Rattus norvegicus”);`



- One value: `my $value = $hash{$key}`
- All keys: `my @keys = keys(%hash)`
- See if something is in a hash: `if(exists $hash{$key}){ … }`

Control Structures

Conditions and Loops

Conditional statements - if

- Basic format:

```
if (condition) {
  run some code here
}
```

- Example:

```
if (length($sequence) > 10 ) {
  print "Seq is longer than 10\n";
}
```

Conditional statements - else

- Basic format:

```
if (condition) {
   run some code here
}else {
   run other code here
}
```

- Example:

```
if (length($sequence) > 10 ) {
     print "Seq is longer than 10\n";
}else{
     print "Seq is not longer than 10\n";
}
```

Conditional statements - elsif

- Basic format:

```
if (condition1) {
   run some code here
}elsif(condition2){
   run something else here
}else {
   run other code here
}
```

- Example:

```
if (length($sequence)  >10 ) {
   print "Seq length is longer than 10\n";
}elsif(length($sequence) == 10){
   print "Seq length is 10\n";
}else{
   print "Seq length is less than 10\n";
}
```

Numerical comparisons

Operator	Meaning	Example
==	Equal to	`if ($a == $b)`
!=	Not equal to	`if ($a != $b)`
>	Greater than	`if ($a > $b)`
<	Less than	`if ($a < $b)`
>=	Greater than or equal to	`if ($a >= $b)`
<=	Less than or equal to	`if ($a <= $b)`

Table modified only slightly from “Perl and Unix to the Rescue”, page 126

String/text comparisons

Operator	Meaning	Example
eq	Equal to	`if ($a eq $b)`
ne	Not equal to	`if ($a ne $b)`
gt	Greater than	`if ($a gt $b)`
lt	Less than	`if ($a lt $b)`
ge	Greater than or equal to	`if ($a ge $b)`
le	Less than or equal to	`if ($a le $b)`

Multiple comparisons

- Put multiple comparisons together in (), separated by:
  - && - and (everything must be true)
  - || - or (at least one must be true)

```
if (($sequence eq “GAATTC”) || ($sequence eq “CTTAAG”)){
    print “Sequence is EcoRI site\n”;
}
```

Loops - while

- Basic format:

```
while (condition){
  do stuff while condition is true
}
```

- Example:

```
my $num = 12;
while ($num <=20){
  print "$num\n"; $num +=2;
}
```

Loops - foreach

- Iterate through a list

```
my @proteins = qw(histone, ubiquitin, actin);
foreach my $protein (@proteins){
  print “$protein\n”;
}

foreach my $i (1 .. length($DNA)) {
    print "Letter $i of the seq is ";
    print substr($DNA, $i-1, 1),"\n";
}
```

Loop control

- `next` – restarts loop
- `last` – terminates loop

```
my $count = 1;
while ($count <= 10) { # repeat for up to ten species
   print "Input species $count abbreviation, or Q to end: ";
   my $species = <>;
   chomp $species;
      if ($species eq "Q") { last; }
      elsif ($species eq "") {
             print "No species entered.\n";
             next; # no grep, counter doesn't change. Ask again.
      }
      system("grep '$species' $blast_out");
      $count = $count + 1;
}
```

Input and Output

Input and Output – File operator

- `<>` - file operator
  - Reads lines of text
  - Remembers how many lines it has seen (`$.`)

- To read in only one line from a file:

```
# if you already specified a file to open, such as
through @ARGV:
# if you didn't specify a file, it will read in a
line of input from the keyboard, equal to <STDIN>
my $line = <>;
```

Input and Output - Filehandles

- Filehandles are special variables used to read or write to files
  - not the same as the file name
  - can open multiple filehandles at once
  - three modes: < read, > write (will overwrite a file if it exists!), >> append
  - must close filehandles once you done operating on them
  - classically written in capital letters

```
open(FH, "<", "filename.txt");
close(FH);
```

Input and Output - Filehandles

- Newer, preferred way to specify filehandles, using scalar variables

```
open(my $input, "<", "in_file.txt") or die "Can't open in_file.txt:
$!\n";

open(my $output, ">", "out_file.txt") or die "Can't open out_file.txt:
$!\n";

while(my $line =<$input>){

      chomp $line;

      my $uc = uc($line);

      print $output "$uc\n";

}

close $input; close $output;
```

Math and functions

Math – operators

**+ - * / % **

```
$y = 6 + 7;    # $y is now 13
$y = $y * 2;  # $y is now 26
$y = $y + 4;   # $y is now 30
$y += 4;       # shortcut for adding, similar operators /=, *=, -=
$y = $y / 10;  # divide by 10
```

Numerical functions

- Functions take arguments, and return a value.
  - Save the returned value to a variable
- abs, int, log, rand, sin, exp
- If worried about operator precedence, use parentheses!

```
$abs = abs(-6); # returns 6

$integer = int(9.7); # returns 9

$y = (4+6) * 7; # force addition before
multiplication
```

Text functions

- Concatenation: `my $cat =“AGG” . “TAC”; # “AGGTAC”`
- Repetition: `print “la” x 40; # prints la 40 times`
- Length: `my $length = length(“xyz”); # yields 3`
- substr(): `my $fox = substr(“quick brown fox”, 12, 3);`
- uc(): `my $upper = uc(”gtcat”);#GTCAT, lc() for lower case`
- join(): `my $joined = join(':', "g", "a", "c"); # g:a:c`
- reverse(): `my $rev = reverse("ACTG"); # GTCA`
- index(): `my $index = index("grandma", "ma"); # 5, -1 if not found`

Regular expressions

Regular expressions

- Fancy pattern searching
- Useful for when you don't know the exact text you want to match
- Requires the =~ Binding operator
  - Matching:

```
print "Found Waldo!\n" if $x =~m/Waldo/; # note that
m can be omitted
```

  - Substitution:

```
$x =~ s/FOO/BAR/g; # global search and replace of
FOO to BAR
```

Regular expression metacharacters

- Anchors: `^` beginning, `$` end
- Characters to escape (\): `^ $ { } ( ) [ ] ? . @ + * / \`
  - `\^` will match a caret, not beginning of string
- Space matching: `\t` tab `\n` newline `\s` space
- Character matching: `\S` non-space `\d` digit `\D` non-digit `\w` word
- `|` or
- `.` Any character but `\n`
- `[^ACT]` any character but A,C, T `[A-Z]` one upper case letter
- `()` save part of the match in special variables - `$1`, `$2`, etc.

Regular expressions – number of matches

- ? – match if it appears or not (0 or 1 times)
- * - match 0 or more times
- + - match 1 or more times
- {n} match n times, {m, n} match minimum m times, maximum n times

	/ab?c/	/ab*c/	/ab+c/
ac	✓	✓	×
abc	✓	✓	✓
abbc	×	✓	✓

Regular expression examples

```
if ($x =~ /^M/){ print “Start codon!\n”; }


if($seq =~m/G{2}[UCAG]/i){ # i is case insensitive
  print “Found a glycine!\n”;
}


my $seq_name = $1 if $line =~m/^>(.*)/;
```

Subroutines

- Create your own Perl functions!
- Reuse code: don't write the same code over and over
- Great for organization, and avoiding bugs
- Call subroutine (pass zero or more arguments), code within subroutine will be executed:

```
&analyze_blast(); # & is optional
```

Subroutine example

```
1  #!/usr/bin/env perl
2  use strict; use warnings;
3
4  my $seq = "CCGGCCGGATGTCTTAGGCGTAGCCGGCCGG";
5  my $gc =  &get_gc_content($seq);
6  print "GC content of sequence: $gc \n";
7
8  sub get_gc_content{
9          my ($seq) = @_;
10         $seq = uc($seq);
11         my $num_g_and_c = $seq =~ tr/CG/CG/;
12         my $length = length($seq);
13         my $gc_content = ($num_g_and_c / $length) * 100;
14         return ($gc_content);
15 }
```

One-liners

- Single line of Perl code executed on command line
- Great for data munging and file manipulation
- Add line numbers to a file

```
perl -wpe 's/^/$.\t/' blah.txt > blah_lines.txt
```

- Global search and replace, retaining original files in file*.bak

```
perl -pi.bak -e 's/FOO/BAR/g' file1.txt file2.txt
```

- Get FASTA ids:

```
perl -wlne 'if (/^>(\S+)/) {print $1}' a.fasta > IDs
```

Example – parsing BLAST tabular output

- Context:
  - BLAST is an alignment tool that can be used with nucleotide or protein query sequence(s).
  - Input sequence(s): *Saccharomyces cerevisiae* sequence
  - Database contains: 12 sequences, including *S. cerevisiae*
- We'll parse the tab-delimited BLAST output in two different ways:
  - blast_parse_array.pl - keep lines in an array, parse for hits over a specified percent identity
  - blast_parse_regex.pl – parse lines using regex, parse for hits from a certain species

BioPerl

- Set of modules for doing bioinformatics in Perl, supported features:
  - Sequence input and output
  - Creating and analyzing alignments
  - Retrieving data from remote databases
  - Converting file formats
- Uses object oriented programming (we don't have time to discuss this today)
- BioPerl is available through module perl/5.24.0 on O2

BioPerl example – renaming seqs in assembly

```
use Bio::SeqIO;
my $in = Bio::SeqIO->new(-file  => $file,  -format => 'fasta');
my $out = Bio::SeqIO->new(-file  => '>redefined.fasta', -format => 'fasta');
my $ct = 1; # Starting numbering scaffolds from 1
while ( my $seqobj = $in->next_seq() ) {
    my $id = $seqobj->display_id; # gets ID from input file. Will move to
description.
    my $desc = $seqobj->description; # Get description field. Should be empty.
    print "Description field is not empty in this sequence:$id\n" if $desc;
    $seqobj->description("Old_ID:$id"); # Description field now has old ID
    my $newid = sprintf(”MYGENOME_scaffold_%06d", $ct); # Modify ID
    my $seq = $seqobj->seq;

    $seqobj->display_id($newid);
    $seqobj->seq($seq);
    $out->write_seq($seqobj);
    $ct++;
}
```

Note on Perl versions

- We've used Perl 5 for this class.
- There is another, newer Perl: Perl 6, which is considered to be a **new language**, and **not a replacement** for Perl 5.
- If you'd like to see the Perl 5 to 6 changes, check out this guide: https://docs.perl6.org/language/5to6-nutshell

Perl Modules

- Other people's code you can reuse!
- Each module is in its own file – ending in `.pm`
  - `.pm` stands for Perl module
- Can access a module with the `use` command (goes at beginning of your script)
  - `use Getopt::Long;`

Where to find available modules?

- CPAN – Comprehensive Perl Archive Network
- Search for modules at http://www.cpan.org/ or https://metacpan.org/ (better searching)
- Install packages with `cpan` or `cpanm`
  - `cpan` – standard tool to install modules, comes with Perl distributions by default
  - `cpanm` – created after `cpan`, less verbose, more user friendly

```
cpan App::cpanminus
cpanm Module::Name
```

Installing Perl modules on O2

- Can install packages yourself using `local::lib`
- Will put packages in the `~/perl5-O2/` directory
- If you're using both O2 and Orchestra, you will need separate personal Perl libraries for each cluster.
- Setup instructions can be found on our Personal Perl Packages page.

Example of installing a module on O2

- Search CPAN to find package
- Assuming `local::lib` is already set up:

```
$ cpanm Text::Fuzzy
```

Module will be installed in `perl5-O2/` in your home directory.

- To use this module, add this at the beginning of your script:

```
use Text::Fuzzy;
```

Documentation

- Use `perldoc` command or reference http://perldoc.perl.org/

`$ perldoc perlsyn`	Syntax
`$ perldoc perldata`	Data types
`$ perldoc perlfunc`	Functions
`$ perldoc -f chomp`	Specific function: chomp
`$ perldoc perlvar`	Predefined variables
`$ perldoc -v @ARGV`	Specific variable: @ARGV
`$ perldoc perlretut`	Regular expression tutorial
`$ perldoc Getopt::Long`	Specific Module

- Use q to exit `perldoc` help pages, arrow keys to maneuver

Resources for after class

- Unix and Perl to the Rescue! by Keith Bradnam and Ian Korf
  - rescuedbycode.com
- Beginning Perl for Bioinformatics by James D. Tisdall
- Learning Perl by Randal L. Schwartz, brian d foy, Tom Phoenix
- Perl One-Liners by Peteris Krumins
  - https://github.com/pkrumins/perl1line.txt/blob/master/perl1line.txt
- Perl introduction: http://perldoc.perl.org/perlintro.html
- bioperl.org, extensive documentation available

Contact information:

HMS Research Computing

http://rc.hms.harvard.edu

rchelp@hms.harvard.edu

Office Hours: Wednesdays 1-3p Gordon Hall 500