1.0.1.8.5

Introduction to Perl
Session 5

- `while` loop
- I/O
- `printf` and `sprintf`

Hash Recap

- hash variables are prefixed by %

```
%fruits = ( red=>"apple", green=>"grape", yellow=>"lemon" );

for $color (keys %fruits) {
  print qq($fruits{$color} is a $color fruit);
}

for $fruit (values %fruits) {
  print qq(For lunch, I brought a $fruit);
}

print "There is a blue fruit" if exists $fruits{blue};
$fruits{purple} = "eggplant";
print "Liar, $fruits{purple} is not a fruit" if exists $fruits{purple};
```

- hashes are used when you need to associate two values together (key and value)
  - you want to keep a count (or other statistic) associated with a key
  - store information centrally that can be recalled by using a string – $options{margin}

Sort Recap

- sorting is performed on a list
  - an optional CODE block specifies how the sort is to be done

```
@nums  = (1..100);
@names = qw(spot fuzz mug flop mac chew wagger);

# default sort is ascending asciibetic
for $name (sort @names) { ... }

# descending asciibetic
for $name (sort {$b cmp $a} @names) { ... }

# ascending numerical
for $num (sort {$a <=> $b} @num) { ... }

# descending numerical
for $num (sort {$b <=> $a} @num) { ... }

# complex - sort by length of name, ascending
for $name (sort {length($a) <=> length($b)} @names) { ... }
```

Shorthand and *crement Operators

- several operations are so frequently performed, that most languages have shorthand versions
  - *man perlop*

```
$a = $a OP $b      →    $a OP= $b

# + - * / short hand
$a = $a + $b       →    $a += $b
$a = $a - $b       →    $a -= $b
$a = $a * $b       →    $a *= $b
$a = $a / $b       →    $a /= $b

# concatenation – add $b to string $a
$a = $a . $b       →    $a .= $b

# post-increment/decrement operator
$a = $a + 1        →    $a++;
$a = $a - 1        →    $a--;

# there is also ++$a and --$a – but we'll skip these for now
```

- do not confuse the match operator ~= here

while Loop

- we've already seen the for loop
  - iterates over items in a list

```
for $num ( @nums ) { ... }

for $key ( keys %hash ) { ... }

for $value ( sort {$b cmp $a} values %hash ) { ... }

for $word ( split(" ",$string) ) { ... }

for $char ( split("",$string) ) { ... }
```

- we now turn to the while loop, which is an iterated if
  - while iterates as long as a specified condition is true

```
# loops as long as CONDITION is true
while (CONDITION) {
  ...
}
```

```
executes once if CONDITION is true
if (CONDITION) {

}
```

while Loop

- while can be used in the same way as for, but this practise is not encouraged
  - I include it here because it nicely illustrates how while works


```
$count = 0;
while ( $count < 5 ) {
  print qq($count);
  # alter the variable used in condition
  $count++;
}

0
1
2
3
4
```

```
# this achieves the same result
for $count (0..4) {
  print qq($count);
}

0
1
2
3
4
```

while Loop

- while loop is used in cases when the appearance of the end-condition cannot be easily predicted

```
($sum,$count) = (0,0);
while ( $sum < 5 ) {
  # alter the variable used in condition
  $sum += rand();
  $count++;
  print qq(Sum of $count numbers is $sum);
}

Sum of 1 numbers is 0.520913321059197
Sum of 2 numbers is 0.644604011438787
Sum of 3 numbers is 1.22890125820413
Sum of 4 numbers is 2.09695924632251
Sum of 5 numbers is 2.73787972517312
Sum of 6 numbers is 3.21515524713323
Sum of 7 numbers is 4.19210437964648
Sum of 8 numbers is 4.62139639491215
Sum of 9 numbers is 5.02754724724218
next check of $sum < 5 fails and while loop is not repeated
```

Loop Control

- you can skip to the next loop iteration without executing the rest of the block using next
  - next if CONDITION;
- you can force termination of the loop using last
  - last if CONDITION;

```
($sum,$count) = (0,0);
while ( $sum < 5 ) {
  $x = rand();
  # skip numbers smaller than 0.5
  next if $x < 0.5;
  $sum += $x;
  $count++;
  print qq(Sum of $count numbers from [0.5,1\) is $sum);
}

Sum of 1 numbers from [0.5,1) is 0.830018592532724
Sum of 2 numbers from [0.5,1) is 1.60385857429355
Sum of 3 numbers from [0.5,1) is 2.58709897752851
...
```

while with shift or pop

- you can destructively iterate across an array
  - by using shift or pop, iteratively whittle an array until there are no elements left

```
@nums = (1..5);
# shift is inside while loop
while ( @nums ) {
  $num = shift @nums;
  print qq(Found $num at front of @nums);
}

Found 1 in front of 2 3 4 5
Found 2 in front of 3 4 5
Found 3 in front of 4 5
Found 4 in front of 5
Found 5 in front of
```


```
@nums = (1..5);
# shift/assignment in condition
while ( $num = shift @nums ) {
  print qq(Found $num at front of @nums);
}

Found 1 in front of 2 3 4 5
Found 2 in front of 3 4 5
Found 3 in front of 4 5
Found 4 in front of 5
Found 5 in front of
```

challenge

- what does this code print?

```
@nums = (0..5);
while ( $num = shift @nums ) {
  print qq(Found $num at front of @nums);
}
```

challenge - answer

- the code prints nothing
  - the first shifted element is 0, which is FALSE and the while block immediately ends
- be careful not to jump out of `while` when encountering zero
  - zero is a perfectly reasonable value (in a file, in an array)
  - be conscious of the potential need for `define` in the condition
  - are you testing for truth (0 is not true) or definition (0 is defined) ?

```
@nums = (0..5);
while ( defined($num = shift @nums) ) {
  print qq(Found $num at front of @nums);
}
```

I/O – Reading from a File

- reading from a file is easy – just like most other things in Perl
- programs which use files typically follow these steps
  - open a file and create a filehandle (special variable type)
  - fetch one line at a time from the file, typically in a `while` loop
    - you parse each line and construct a data structure that holds this information
  - close the file when EOF (end of file) is encountered
    - the EOF evaluates to FALSE, which nicely terminates any `while` loop
  - process data
  - optionally, write data out to another file

- we will see how to
  - read from a file
  - write to a file
  - format strings with `printf`/`sprintf`

Reading from a File

- reading from a file is done in three steps

```
# 1. open the file, creating a filehandle (conventionally capitalized)
open(FH,$file);

# 2. fetch next line from the file (via filehandle) using line operator <>
while( $line = <FH> ) {
  ...
}

# 3. close the filehandle (not strictly necessary, but good housekeeping)
close(FH);
```

- capitalize your filehandles (convention)
  - FILE, FH, HANDLE but not file, fh, or handle
  - you can have multiple handles
  - eventually you will use modules (`IO::File`) that abstract raw filehandles
  - *I lied, there is another variable type*

Reading from a File

```
> cat file1.txt
0
1
2
3
4
```

```
open(FH,"file1.txt");
while( $line = <FH> ) {
  print $line;
}
close(FH);

0
1
2
3
4
```

- the <> operator (slurp operator) returns the next line
  - includes the trailing newline
- the test within the `while()` loop when used with <> is implicitly a defined test
  - a file with a "0" and trailing newline "0\n" will evaluate to true because of the trailing newline
  - a file with a trailing "0" without a newline will evaluate to false, which is inconvenient and thus Perl applies defined
  - loop ends when <> returns EOF after the last line
- <> actually returns the next record in the file
  - default record terminator is "\n", thus you get a line at a time
  - you can change the record terminator and modify the behaviour of <> (*here be dragons*)

chomping lines

```
> cat file2.txt
a b c
d  e   f
0  1     2
3 4
5

open(FH,"file2.txt");
while($line = <FH>) {
  # remove trailing newline in $line
  chomp $line;
  # split line at whitespace (i.e. into words)
  @words = split(" ",$line);
  # concatenate words together using ":"
  print join(":",@words);
}
close(FH);

a:b:c
d:e:f
0:1:2
3:4
5
```

- chomp safely removes the trailing newline in a string
  - it does nothing if a newline is not present
- `@list=split(" ",$line)` tokenizes the line into words
  - at whitespace (spaces or tabs)

File Analysis Script

- we will create a script that performs the following
  - reads from a file
  - reports the number of words on each line
  - reports the total number of lines and words in the file
  - reports the average number of words per line
  - returns the 5 most common words
  - returns the 5 longest words

Step 1 – parsing the file

```
# keep count of each word in a hash
%words = ();
# keep number of words per line in an array
@wordcount = ();

open(FH,“sherlock.txt");
while($line = <FH>) {
  chomp $line;
  # split line at a run (one or more) non-word characters : \W is the opposite of \w
  @words = split(/\W+/,$line);
  # iterate through all words in the line
  $wordcount = 0;
  for $word (@words) {
    # accept only words which have a letter character (e.g. no numbers)
    if($word =~ /[a-z]/i) { # /i is for case-insensitive match
       # increment count for this word
       $words{$word}++;
       $wordcount++;
    }
  }
  # add number of passed words in this line to the array
  push @wordcount, $wordcount;
}

close(FH);
```

Step 2 – reporting word statistics

```
$wordcount_total = 0;
for $i (0..@wordcount-1) {
  # maintain count of all words seen
  $wordcount_total += $wordcount[$i];
  # report on words on this line
  print qq(line $i had $wordcount[$i] words);
}

# report on word count statistics
print qq(saw ),scalar(@wordcount),qq( lines in file);
print qq(saw $wordcount_total words in file);
print qq(average words/line ),$wordcount_total/@wordcount;

# create sorted word lists – by frequency and length
@words_common = sort { $words{$b} <=> $words{$a} } keys %words;
@words_length = sort { length($b) <=> length($a) } keys %words;

for $i (0..4) {
  print qq(common word $i $words_common[$i]);
}
for $i (0..4) {
  print qq(longest word $i $words_length[$i]);
}
```

```
line 0 had 9 words
line 1 had 0 words
line 2 had 0 words
...
line 12649 had 0 words
line 12650 had 0 words
line 12651 had 0 words


saw 12652 lines in file
saw 105999 words in file
average words/line 8.3780429971546

common word 0 the
common word 1 I
common word 2 and
common word 3 to
common word 4 of
longest word 0 disproportionately
longest word 1 indistinguishable
longest word 2 conventionalities
longest word 3 scissorsgrinder
longest word 4 inconsequential
```

Reading FASTA files

- FASTA file is a simple sequence format
  - first line starts with “>” and contains a header
  - first word in header is referred to as the ID
  - sequence follows, usually 50-80 bp per line

```
>gi|4878025|gb|U80929.2|CVU80929 Cloning vector pBACe3.6, complete sequence
GATCCGCGGAATTCGAGCTCACGCGTACTGATGCATGATCCGGGTTTAAACCCAGTACTCTAGATCCTCT
AGAGTCGACCTGCAGGCATGCAAGCTTGGCGTAATCATGGTCATAGCTGTTTCCTGTGTGAAATTGTTAT
CCGCTCACAATTCCACACAACATACGAGCCGGAAGCATAAAGTGTAAAGCCTGGGGTGCCTAATGAGTGA
GCTAACTCACATTAATTGCGTTGCGCTCACTGCCCGCTTTCCAGTCGGGAAACCTGTCGTGCCAGCTGCA
TTAATGAATCGGCCAACGCGCGGGGAGAGGCGGTTTGCGTATTGGGCGCTCTTCCGCTTCCTCGCTCACT
...
ACTTCGTATAGTATACATTATACGAAGTTATCTAGTAGACTTAATTAAGGATCGATCCGGCGCGCCAATA
GTCATGCCCCGCGCCCACCGGAAGGAGCTGACTGGGTTGAAGGCTCTCAAGGGCATCGGTCGAGCTTGAC
ATTGTAGGACTATATTGCTCTAATAAATTTGCGGCCGCTAATACGACTCACTATAGGGAGAG
```

- there are modules that help you process FASTA files
- let’s write a script to read a FASTA file and produce statistics

Step 1 – reading FASTA file

```
%bp = ();
# open human chr22 assembly
open(FH,"/home/martink/work/ucsc/hg18/fasta/chr22.fa");
while($line = <FH>) {
  chomp $line;
  # skip header
  next if $line =~ /^>/;
  # we're in the sequence
  # store count for each unique bp
  for $bp (split("",$line)) {
    $bp{$bp}++;
  }
}
```

- the ^ in regexps is an anchor which matches start of line
  - /^hello/ matches hello at the start of a line
- the $ in regexps is an anchor which matches end of line
  - /goodbye$/ matches goodbye at the end of a line
- challenge – what does /^$/ match?

Step 2 – processing base pair types

```
%bpstats = ();
for $bp (keys %bp) {
  # print count for each bp type seen
  print qq($bp $bp{$bp});
  # keep independent count of different bp types
  if ($bp =~ /[atcg]/) {
      $bpstats{repeat}  += $bp{$bp};
  } elsif ($bp =~ /n/i) {
      $bpstats{padding} += $bp{$bp};
  } elsif ($bp =~ /[GC]/) {
      $bpstats{gc}      += $bp{$bp};
  } elsif ($bp =~ /[AT]/) {
      $bpstats{at}      += $bp{$bp};
  } else {
      $bpstats{unk}     += $bp{$bp};
  }
}

# return count of bps, by category
for $statistic (sort {$bpstats{$b} <=> $bpstats{$a}} keys %bpstats) {
    print qq($statistic $bpstats{$statistic});
}
```

```
a 4510978
A 4555927
c 3824338
C 4516771
t 4480960
T 4544042
n 17
N 14789904
g 3813956
G 4517817

repeat 16630232
padding 14789921
at 9099969
gc 9034588
```

Writing to a File

- the easiest way to write to a file is to redirect the output of your script to a file
  - anything printed to STDOUT will be sent to a file
  - anything printed to STDERR will be sent to the screen

```
% my_script.pl > file.txt
```

  - to redirect both STDOUT and STDERR to a file,

```
% my_script.pl &> file.txt
```

  - to redirect to different files,

```
% my_script.pl > file.txt 2> file.err.txt
```

Writing to a File

- sometimes you need to write to a file from your script
- open the file with `open()` but prefix filename with `>` or `>>`
  - `>file` create and overwrite if necessary
  - `>>file` append
- pass the filehandle as the first argument to print
  - `print FH $num`

```
open(FH,">file.txt");

for $num (@nums) {
  print FH $num,"\n";
}

close(FH);
```

- **no comma** between filehandle and arguments to print
  - print FH, $num1, num2;

Creating a Random FASTA file

- let's create a random 100,000 bp FASTA file

```
@bp = qw(a t g c A T G C);
$sequence = "";
for (1..100000) {
    # $array[rand(@array)] idiom - randomly samples array
    $sequence .= $bp[rand(@bp)];
}

open(FH,">randseq.fa");
print FH ">random_sequence\n";
# 4-argument substr returns 70 characters and replaces them with empty string
while( $line = substr($sequence,0,70,"") ) {
    print FH $line,"\n";
}
close(FH);
```

```
>random_sequence
CCCGagttcAtGCGTCcTcATAAtgTTaGAGTcGAAtTTTgCctTaatTAGcagAcatcGTgAttaTcGg
aatctCAgagCCTCttcgcGtttTggTaTcgGcAgTcGaAaCcGCTagacatTgGaActgCcacagtAtt
...
cGcaACctCCacaAcTGgGtGgGTacagtCATGaTgCtAGtTgttTCCaTaGaGcagAcAcCttCGcCaa
TtTCgGTtTTCGACTCCCAccCgTagAAtAACGtCaCTgT
```

Formatted Output

- it is often desirable to prettify output for better readability
  - pad strings to fixed number of characters
  - specify number of decimals in a string
- `printf` is used to output a prettified string
- `sprintf` is used to generate a prettified string (which may be printed)

```
printf FORMAT_STRING,LIST;

$formatted_string = sprintf FORMAT_STRING,LIST;
```

- the `FORMAT_STRING` specifies how the elements in the `LIST` are to be presented
  - contains special entries like `%s`, `%d`, `%f` used to format `LIST` elements

printf

```
@x = qw(0 1 1.0 1.6234);

# truncated, not rounded
printf "%d %d %d %d\n",@x;
0 1 1 1

# default 6 decimals
printf "%f %f %f %f\n",@x;
0.000000 1.000000 1.000000 1.623400

# each field length is 10
printf "%10d %10d %10f %10f\n",@x;
         0          1   1.000000   1.623400

# fix decimal places for floats
printf "%10d %10d %10.3f %10.3f\n",@x;
         0          1      1.000      1.623

# left justify first two fields
printf "%-10d %-10d %10.3f %10.3f\n",@x;
0          1                1.000      1.623

# and now zero padding
printf "%-010d %010d %010.3f %010.3f\n",@x;
0          0000000001 000001.000 000001.623
```

- %d – integer output
- %f – float output
- %s – string output

- %Nx – field length N (%3d)
- %.Dx – D decimal digits, where applicable (%.2f)
- %N.Dx – field length N with D decimal digits, where applicable (%5.2f)
- %0x – 0-pad (%05.2f)

- %-x – left justify (%-05.2f)

sprintf

- let's create a script that produces the following
  - a random number
  - a 5-decimal truncated version of it and its square
  - a digit map (number of times each digit 0-9 seen in the number)

```
open(FH,">data.txt");
for $i (1..100) {
  $x = rand();
  @digits = ();
  for $char (split("",$x)) {
      # count the number of times digit $char is seen
      # e.g. increment $digits[5] everytime 5 is seen
      $digits[$char]++ if $char =~ /\d/;
  }
  $line = sprintf("line %3d rand %20s trunc %.5f trunc^2 %.5f digitmap %d%d%d%d%d%d%d%d%d%d",
                  $i, $x, $x, $x**2, @digits);
  print FH $line,"\n";
}
close(FH);

line   1 rand    0.929494368378073 trunc 0.92949 trunc^2 0.86396 digitmap 2013201223
line   2 rand    0.903590672183782 trunc 0.90359 trunc^2 0.81648 digitmap 3122011222
...
```

Example - Filtering Files

- let's take all end sequence alignments of human clones and report the average clone sizes for different groups of clones, defined by regular expressions
  - one file contains the data (coordinates)
  - another file contains the filters (regexps) used to process the data
- coordinates defined in a file like this

```
M2131014 CTD-2131014 3 80809618 80926601
M2131015 CTD-2131015 6 121610096 121675696
...
```

- clone groups defined by file containing regular expressions

```
#comment
A01$
10$
^N
^D
^.0001
```

Step 1 – Reading coordinates and categories


```
%clonesize = ();
open(FH,"/home/martink/work/ucsc/hg17/bes/bacend.parsed.txt");
while($line = <FH>) {
    chomp $line;
    # M2131O15 CTD-2131O15 6 121610096 121675696
    # assigning to undef effectively skips the field
    ($clone,undef,$chr,$start,$end) = split(" ",$line);
    $size = $end - $start + 1;
    # keep track of size of each clone
    $clonesize{$clone} = $size;
}
close(FH);

# make an array of the regular expressions
@rx = ();
open(RX,"rx.txt");
while($line = <RX>) {
    chomp $line;
    # skip past comment lines
    next if $line =~ /^\s*#/;
    push @rx, $line;
}
close(RX);
```

Step 2 – Processing clones

```
%sum = ();
%count = ();
# check each clone whose size we know
for $clone (keys %clonesize) {
  # iterate through each regular expression
  for $rx (@rx) {
    # if the clone matches then keep track of total size/count for this category
    if($clone =~ /$rx/) {
      $sum{$rx}   += $clonesize{$clone};
      $count{$rx} ++;
      # last is a flow-control key word which terminates the innermost enclosing loop
      last;
    }
  }
}
for $rx (sort keys %sum) {
  printf("group %10s num %6d avgsize %8.1f\n",
         $rx, $count{$rx}, $sum{$rx}/$count{$rx});
}
```

```
group        10$ num   8502 avgsize   148126.2
group       A01$ num    389 avgsize   140548.0
group     ^.0001 num    195 avgsize   163235.3
group         ^D num  39519 avgsize   138941.2
group         ^N num  99252 avgsize   170593.5
```

1.0.8.1.5

Introduction to Perl
Session 5

- you now know
  - `while` loop
  - reading from a file
  - writing to a file
  - `printf/sprintf`
- next time
  - subroutines
  - introduction to special variables
  - `$_` and friends