Perl Module_1 Review

#!/usr/bin/perl
$/ = undef;

- # - Commenting out code (indicating directives)
- **!** - Indicates what type of program is running, in our case, a perl script (this would not be necessary in DOS, if one is using the correct path to the Perl Interpreter and the script.

- Most all perl statements end in semicolon (;)

$scalar = <FILEHANDLE> (the input file is entered into memory) (Angle brackets are Perl syntax)
Scalars (variables for strings and numbers) can hold $count, $file, $defline and there is no need to declare scalars as in C or C++
Variable interpolation - is used for statements containing variables in double-quotes, which are substituted by their actual value in Perl run-time.

- **$/** (**Input record separator**) - Is a system variable which when equal to undefined, allows perl not to continue reading input file after the first line separator (\n)

tr/// operator: tr/old/new
$scalar = ~ tr/old/new/; ($scalar also keeps tracks of number of translations performed)
$newvariable = ($scalar =~ tr/old/new/); (parenthesis to indicate operation taking place first)
$newvariable = ($scalar =~ tr/old//); (which only counts and does not replace)
Examples of \c, \d and \s modifiers:
$dnaseq = "AATTGGCCTG";
$count = ($dnaseq =~ tr/A//c); counts everything apart from A, similarly
$count = ($dnaseq =~ tr/A/x/c); counts and replaces everything apart from A to x
$dnaseq =~ tr/A//d; deletes all A's
\s option shortens the output of the tr/// operation and prints a single character for every repetitive adjacent occurrence of the same character

s/// operator: s/old/new
$scalar =~ s/(choice1|choice2|choice3…)//; (only substitutes the FIRST occurrence of the specified term
need to add a 'g' modifier to make substitutions global
$scalar =~ s/atg/start/g;

=~ *matching operator* that tries to look for an expression similar to one specified by the user on the right hand side of the eq.

print is an in-built Perl function that prints the text argument supplied to it.
print "=" x 70 will print '=' seventy times
printf (type of number) (type of output desired)
floating-point numbers or integer in decimal format
field specifiers for floaters is %f (%.1f prints one digit after decimal)
field specifiers for decimals is %d
eg. Printf "GC content: %.1f%\n", $GC;

Error checking code: die is the syntax to abort script followed by the special variable $! Which prints out the error message, as the script exists.

chop – removes any last character from an input string whether it is a new-line (\n) or not
chomp – only removes new-line (\n) characters, if present.
Chomp ($input);

Summary of **Perl mathematical operators**:

Operator	Symbol	Example
Numerical assignment	=	$x = 3;

Equality comparision	= =	$x = = 3;
Not Equal To	!=	$x != 3;
Less than	<	$x < 3;
Greater than	>	$x > 3;
Less than or equal to	<=	$x <= 3;
Greater than or equal to	>=	$x >= 3;
Addition	+	$z = $x + $y;
Subtraction	-	$z = $x - $y;
Multiplication	*	$z = $x * $y;
Division	/	$z = $x / $y;
Exponentiation	**	$z = $x ** $y;
Modulus (Remainder)	%	$z = $x % $y;
+=	**$x += 5;**	$x = $x +5;
++	**$x++;**	(Increments x by 1)
-=	**$x -= 5;**	$x = $x – 5;
*=	**$x *=5;**	$x = $x * 5;
/=	**$x /=5;**	$x = $x / 5;
%=	**$x %=5;**	$x = $x %5;
**=	**$x **=5**	$x = $x ** 5;

Perl Web resources:
Official Perl homepage: http://www.perl.com
Bioperl Project: http://www.bioperl.com/
CPAN: http://www.cpan.org/
The Perl Journal: http://www.sysadminmag.com/tpj/
Perl documentation: http://www.perldoc.com/
The Perl Archive: http://www.perlarchive.com/
Perl Mongers: http://www.perl.org/
Entrez: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi (ftp ncbi.nlm.nih.gov)
Brookhaven National Laboratory: http://www.bnl.gov/
DDBJ: DNA Data Bank of Japan: http://www.ddbj.nig.ac.jp/Welcome-e.html
EMBL: European Molecular Biology Laboratory: http://www.embl-heidelberg.de/

Array Variable is declared with the @ type identifier, and array data is declared by enclosing the values in parenthesis and separating them by commas (it stores a set of scalars). Numericals and strings within quotes can be stored.
Array (and scalar) cannot have names starting with characters such as dashes, underscores, dots or numbers

$array[i] = constant (I = index, which, by default, starts from 0, unless we set $[= 1)
@values = ($values[0], $values[1],…)

$file = ‘c:\jobs\science_sept14.txt’; # dos uses back slashes
$file = ‘c:/jobs/science_sept14.txt’; # unix uses back slashes

Array type:
@lines [array of lines in a file to be parsed]
@scores [array of Blast scores]
@file [array holding file contents] or
@ARGV [special variable that holds command line arguments] ie. $scalar = ARGV [0] takes in first argument and $scalar2 = ARGV [1] takes in the second argument (could be used in place of GetOpt::Long module.

To copy array: @array1 = @array2;
To create empty array or erase existing array: @array1 ***= ();***
Populating arrays with sequential data using range operator ***(..)****:*

```
@numbers = (1..10);
@alphabets = (‘a’ .. ‘z’);
@alphanumeric = (‘a’ .. ‘z’, 1..100);
```

*Quote Word Function **qw**:*

@enzymes = ("EcoRI", "BamHI", "HindIII"); is the same as

@enzymes = **qw**(EcoRI BamHI HindIII); #comma separators are not needed

Size of an array:

$size = @array; or Counting array elements with the *scalar* function –

$size = scalar(@array);

Accessing the last element in an array:

$array[@array-1]; or with the $#notation

$array[$#@array]; or accessing the last element by negative index

$array[-1];

Adding elements to the end of an array:

push (@array, "$scalar"); #adds to the end of the existing elements in the array

unshift (@array, "$scalar"); #adds elements to the beginning of an array

shift (@array); #removes the first element from an array

pop (@array); #removes elements from the end of an array

Array Slicing:

@slice = @enzyme [0,1]; #will remove elements 0 & 1 from enzyme array to create a new array called slice.

@splice = @enzyme [0..2]; #use of '..' range operator

@splice = @enzyme [@range]; #where @range = (0..2); where range is specified as an array

splice ([*array to splice*], [*offset*], [*length*], [*new elements*]); #where

offset = starting index from where elements are to be removed

length = number of elements from the offset number to be removed

list = list of values to replace the removed elements with; elements deleted if this is left blank

Sorting Arrays:

@newlist = **sort {$a cmp $b}** @oldlist; #[$a, $b - internal variables for lexical sort function]

@newlist = sort @oldlist; #[without the cmp operator as sorting is lexical by default]

@newlist = sort **{$a < = > $b}** @oldlist; # [< = > comparison operator for numerical sorting]

@reverse = **reverse** (@array); # for reversing a sort

@reversesort = **reverse sort** (@enzymes); # combining reverse and sort

Split syntax:

@array = **split**(/regex/, $scalar); eg of a regex within a function as well!

(A descriptor used in the split() function to process data, itself, does not appear in the saved records)

split (/delimiter/, $string); # syntax for split function

Another example of split function is:

$string = "gene:helicase";

($x, $y) = split(/:/, $string); # where $x = gene and $y = helicase

Creating Strings from Arrays:

join() creates a string out of an array where each individual string is joined by the specified delimiter

$genes = join(" ", @genes);

The delimiter is specified in quotes in join() while it is specified in // in split(). Since quotes do not indicate a regex, we cannot use "\s" with join() as we do with split() to indicate single space delimiters.

We can also use the **chop** operator to process information in a given array: **chop(@array);**

Regex:

Perl regex's search for defined patterns and performed operations on them as specified by user

Regex in Perl are enclosed within forward slashes: /regex/ and may contain strings or variables

A matching|binding operator =~ looks for exact match to the specified patter; to perform a reverse operation, ie when a match is not found – the !~ operator is used instead of the =~.

Syntax for regex:

$regex =~ /(first) (second) (third)…(nth)/;

$first = $1; ## $first = 'first'

$second = $2; ## $second ='second' etc. etc.

any two identical characters can be used to specify a delimiter. The only difference is with delimiters other than /, the pattern matching operator m must be specified. $scalar =~ m!search_term!;

Special Character

+: this operator is used to match one or more preceding characters (greedy operator) eg /ez+/ will match ez, ezz, ezzz, ezzzz and so on and will return only the maximum one is. Ezzzz
The operator ? used in conjuction with the greedy operator + will limit the match to the first occurrence.
*****: limits the matching of the preceding character to zero or more occurrences of the preceding character. Regex /ez*/ is the same as /ez*?/
?: limits the matching of the preceding character to zero or **at most** one occurrence of the preceding character
\s: matches a single space or [\n\t\r\f]
\S: matches any single non whitespace [is defined as a space, new-line character, tab, carriage return or a form-feed] or [^\n\t\r\f]
\s+: matches one or more spaces
\d: matches a single digit same as [0-9]
\d+: matches one or more digits
to test for the presence of a string or variable in a regex, we place it in parenthesis and place '?' outside. If the string is present zero or one time, then $1 = 0 else $1 = 'string'.
\D: matches any single non-digit [^0-9]
\w: matches any single word or [_0-9a-zA-Z]
\W: matches any single non-word or [^_0-9a-zA-Z]
[] brackets: specifies a range of characters to match
[0123456789] or **[0-9]:** matches any single digit
[a-zA-Z]: matches any single upper or lower case letter
[A-Z0-9]: matches any single upper case letter or digit
[0-9_\-]: matches any single digit, underscore or a dash
[^0-9]: carat immediately after the left bracket matches the absence of particular set specified
[A-Za-z0-9]+: will match something like OSJNBa0058E19
Escape sequences: are used to escape out characters such as;), (, *, + etc. et. \(or *
Use double backslashes to escape a backslash eg. If ($string =~ /\\/) {do something};
Anything enclosed within **\Q** & **\E** escape sequences is treated as a regular text character eg. /\QCa++\E/

Match quantifiers:

+ [matches one or more instance of pattern] +{1, }
***** [matches zero or more instance of pattern] *{0, }
? [matches zero or one instance of pattern] ?{0,1}
can be used with any pattern to match (eg ez{3,5} etc)

Pattern Anchors:

^ or **\A** [matches at beginning of a string] eg. Regex – ($seq =~ s/^\s+//) will remove all spaces from beginning of string
$ or **\Z** [matches at end of string]
\b [matches at beginning or end of word]
\B [matches only inside a word]

Metacharacters (pattern modifier operators):

. (dot) matches single character
.+ (dot plus) matches one or more characters
s (for single) modifier enables Perl to treat expressions that spill over multiple-lines as one continuous line eg. If ($job =~ /$search_term.+URL:(.+)/s);
i (switch) allows us to include case-insensitive search in a regex
g: enables global substitutions
|: enables matching a list of patterns in ()
m: treats patterns as multiple lines (modifies how ^ & $ behave in regex for multi-line strings)
x: allows the addition of spaces and makes it easy to construct a regex (used with pattern comments)

e: forces the replacement string in the substitute function to be treated as an expression that is evaluated before replacement. Eg.

$string = "10 20 30 40 50";
$string =~ s/(\d+)/$& * 10/g**e**;
$string becomes: 100 200 300 400 500

Pattern system variables: (when a pattern is matched successfully)

$& returns the entire matched string
$+ returns the pattern that the last bracket matched
$` returns everything before the matched string
$' returns everything after the matched string

Conditional matching operators:

?=: conditional positive matching (eg. $cds =~ /complement(?=(.+))/
?!: conditional negative matching (eg. /gene(?!=complement)/ will query for pattern containing 'gene=' not followed by 'complement'

Perl Control Modifiers:

General Syntax: modifier (condition) {statement block}

Foreach loop: allows you to access each element of an array in succession.

foreach $element(@array) {do something}; #only code in curly brackets is used for foreach loop

If loop syntax:

```
if ($scalar =~ /search_term/) {do something;}
```

If – else syntax:

```
if (evaluate_condition) {
        if_condition_true_execute_if_block;
        if_condition_false_go_to_else_clause;
}
else {
        execute_else_block;
}
```

If – elsif syntax:

```
if (evaluate_condition) {
        if_condition_true_execute_if_block;
        if_condition_false_go_to_elsif_clause;
}
elsif (evaluate_condition) {
        if_condition_true_execute_elsif_block;
        if_condition_false_go_to_next_elsif_clause;
        iterate_over_all_elsif_conditions;
}
```

If – elsif – else syntax:

```
if (evaluate_condition) {
        if_condition_true_execute_if_block;
        if_condition_false_go_to_elsif_clause;
}
elsif (evaluate_condition) {
        if_condition_true_execute_elsif_block;
        if_condition_false_go_to_next_elsif_clause;
        if_all_elsif_conditions_false_go_to_else_clause;
}
else {
        execute_else_block;
}
```

- Unlike *if*, neither *else* nor *elsif* can be used alone!
- The *else* block executes only if all the preceding *if* or *elsif* conditions evaluate to false.

- Whenever *else* is present in an *if* statement, it is always specified last. *It doesn't have a conditional expression associated with it*.

Unless Modifier:
Syntax: **unless (**evaluate_condition**) {**execute_block**;}**
Unless is the opposite of if and is executed only if a condition is *not met!*
Example: die ("Error opening $file: $!\n") **unless** (open(IN, $file)); is the same as
die ("Error opening $file: $!\n") **if (!)** (open(IN, $file)); #negation operator

Control operator: **next unless** – allows us to only search an array element if it match a particular condition – eg next unless $scalar =~ /$search_term/;

While Modifier:
Syntax: **while (**condition_is_true**) {** execute_block**;}**
Example:

```
while ($line = <IN>){
        if ($line =~ /PlyA/) {print "$line\n";} #can be also written with default variable $_

while ($line = <IN){
        if ($_ =~ /PlyA/) {print "$line\n";}
```

Until Modifier:
Syntax: **until (**condition_is_false**) {**execute_block**;}** #is the opposite of while and executes while the conditional expression is false; or to rephrase it…

```
Up_to_the_time_that (condition_is_false) {execute_block;}
When (condition_is_true) {stop;}
```

Eg: **until** ($input = = $password) {print "Wrong password\n";}

For Modifier:
Syntax: **for (**initial state**;** condition**;** change_state**) {**execute_block**;}** #like while but complex condition

Initial_state	[Initialization of variables]
Condition	[the test condition]
Change_state	[increment/decrement variable]

Note: The incremental statement in a for loop does not have a terminating semi-colon
Example:
for ($gene_number = 1**,** $exon_number = 1**;** $gene_number <11**;** $gene_number++**,** $exon_number++**)**
{ execute_block**; }**

Last, Next and Redo Modifiers:
last allows us to exit out of loops when a required condition is met
next allows us to skip over a iteration when a specific condition is encountered
redo allows us to restart an interation until the condition is met Eg for passwords

---

Lexical or Static scoping of variables:

- The keyword ***my*** is a way of initialising variables. This allows the scalar declared by *my* to be reset every time it's assigned a new value through a loop.
- The **my** variable is private and is visible only in the code block in which it is declared.

Example:

```
foreach $line(@lines) {
        my $size;
        $count++;
        @exons = split (/\n/, $line);
        print "$count]\t";

foreach $exon (@exon) {
        if ($exon =~ /(Init|Intr|Term|Sngl)\s+(\+|\-)\s+\d+\s+\d+\s+(\d+)/) {
                $size += "$3";
        } #end if
} #end foreach
```

- Here, the value of $size, is not cumulative, because of re-initializing of $size by *my*!

Dynamic Scoping:

- Is done by using the ***local*** keyword to declare variable

---

Modules are packets of code that impart additional functionality to your programs. They have in-built methods that provide the means to carry out specialized tasks.
Getopt::Long -> enables script to parse command line arguments (uses a file called Getopt/Long.pm on system

Syntax: use module_name;
Function is **GetOptions()** or
(GetOptions("f|filename=s" => \$file)); the f is the flag whose value provided on the command line is transposed into the $file variable. Flag can be provided as (–f or --filename)
(GetOptions("f|filename=**s**" => \$file)); for passing strings
(GetOptions("v|value=**i**" => \$value)); for passing integers
(GetOptions("p|price=**f**" => \$price)); for passing real number arguments
(GetOptions("f|filename=s" => \$file, "s|search=s" => \$search_terms)); syntax for multiple options

A ***hash*** is designated with a **%** symbol and consists of data that are organized as key-value pairs separated by a delineator. The delineator could be => or a simple comma. GetOptions() is a storing argument that needs to run in the form of a hash where the flag 'filename' is the data key and the value of the key is \$file.

Getopt::Std -> is the module that preceded getopt std, except here the arguments are bundled together, instead of being specified individually. The function **getopt()** creats a **global variable** identified by the exact letter the user has hard-coded in the function. The global variables then are **$opt_f** or **$opt_s** for arguments f & s respectively. The program can be run using single letter flags, but not long names.
Syntax for Getopt::Long flags:

F|flag = s	Mandatory string argument
F|flag : s	Optional string argument
F|flag = I	Mandatory integer argument
F|flag : I	Optional integer argument
F|flag = f	Mandatory real number argument
F|flag : f	Optional real number argument

LWP::Simple Module: or libwww-perl, is a set of Perl modules that provide methods to access and retieve information from web pages.
Function ***get()*** supplied by the LWP::Simple modules requires only the URL information.
Syntax: $page = **get** ($url);

 is HTML equivalent of line-breaks (ie. \n)

File::Basename: provides the functions ***dirname()*** and ***basename()*** to parse the file and the directory portions of a given path. The function ***fileparse()*** can be used to parse file extensions using the regex: '**\..***'. Examples include:

$dirname = **dirname** ($filename);
$basename = **basename** ($filename);
($filename, $dirname, $extn) = **fileparse** ($filename, '\..*'); where
'\..*' could be '\.bak' or '\.pl' or '\.(pl|bak)'

Cwd Module: provides the ***cwd*** function to obtained the current working directory

use **cwd**;
$current_dir = cwd;

File I/O:
Syntax for library function open:
open (FILEHANDLE, $scalar); #the argument to specify the FILEHANDLE – actually a file variable – is written in upper case to readily differentiate it from other file variables used in the script.
Read: open(READ, '**<**filename'); same as
open(READ, 'filename');

Write: open(WRITE, '**>**filename');

Append: open(APPEND, '>>filename');

Read+Write: open(RW, '+<filename');

- All filehandles are arbitrary!

- Command 'close' can be used to close open files: eg **close** (FILEHANDLE);

File test operators:
Example of an operator: if (**-e** 'filename') {print "filename exists!\n";}
If (**!**(**-e** 'filename')) {open ">filename";}
Other file test operators are:

-d	If filename is a directory
-e	If file exists
-s	If file is non-empty
-z	If file is empty
-r	If file is a readable file
-f	If file is a plain file
-T	If file is a Text file
-w	If file is writable
-x	If file is executable
-B	If file is binary

Accessing files with <>:
There are a number of ways to access the contents of a file. The **file input operator** (**<>**) eg:

```
while (<IN>) {print "$_\n";}
```

The while loop causes the file associated with the IN file handle to be read **one line at a time**; it terminates when the end-of-file (**EOF**) is encountered, at which time **<> returns false**.

Accessing files with @ARGV variable:
Takes in arguments from the command line and processes them inside script. Eg.

```
while ($file = shift @ARGV) { print "$file\n"; }
```

The name of the program that processes command line arguments is stored in special variable: **$0**
$#ARGV is used the calculate the number of elements in the array ARGV (or # of command line arguments)
When combined:

```
if($#ARGV < 2) {print "Usage: $0 file_name search-term";}
```

or it can be re-written as:

```
if($#ARGV < 2) {
        print << "USAGE";
        $0: Script to search a file for a given key word
        Usage: $0 file_name search_term
        Where:
                file_name               : Input file
                search_term             : Input key-word
USAGE
}
```

Here the print << "USAGE" prompt causes everything to be printed upto the flag USAGE, if the condition is met!

Deleting Files: the ***unlink*** function is used to delete files in Perl. Syntax: **unlink** $filename

Opening directories: ***opendir*** (DIR, $dir); [DIR is the directory handle] and can be used for error checking such as: die "Error opening $dir: $!\n" unless opendir(DIR, $dir);

Reading directories: **readdir**(DIR);

When the entire contents of a directory need to be manipulated, it is convenient to use a while loop: **while**(defined($file = **readdir**(DIR))){…} **#takes directory handle as argument and not pathname.**

Create new directory: **mkdir** ($dir); or to make subdirectory:

mkdir("$dir\\$subdir");

Changing directory: **chdir**($dir);

Removing directory: **rmdir**($dir);

Closing directory: **closedir**(DIR); [**#takes directory handle as argument and not pathname]**

System function: The above commands can also be run as a system command. This function executes any statement as if it's been executed on the command line.

Moving files: DOS: system("move c:\\perl\genscan.txt c:\\perl\\genscan");
UNIX: system("**mv** /home/sequences/genscan.txt /home/genscan");

Deleting files: DOS: system("del c:\\perl\\genscan.txt");
UNIX: system("**rm** /home/genscan/genscan.txt");

Subroutines:

Subroutine is a portion of code that resides in its own code block which is defined by the **{ }**. They have their own definition and can be "called in" to execute. They are defined and called with the **sub** keyword. A subroutine can also be called in with **ampersand (&).**

Subroutines are useful for making code more modular and allows re-use of code

Subroutine parameter: constitute a mechanism to pass arguments or values to subroutines. Arguments passed are seen by subroutine as list variables or values in special variable **@_** (contained in the order in which they are passed). These arguments can be accessed like elements of an array:

foreach $parameter(@_) {print $parameter;}

Where $_[0], $_[1], $_[2] is equal to each element.

The special variable can be broken into scalars as such:

($name, $strand, $start, $stop) = @_;

print "$name $start $stop\n";

Other Perl built-in functions:

Index function: returns the position of the first occurrence of SUBSTR in STR at or after POSITION.

$variable = **index(STR, SUBSTR, POSITION)** or

$variable = index(STR, SUBSTR) where it starts searching from the beginning of STR

$[Variable: The subscript for the first character of STR is zero, however, this can be changed by setting the $[variable.

$dna = "AA**GAATTC**CCGAATTC";

subscript**($[= 0)** = 0 1 **2** 3 4 5…

subscript**($[= 1)** = 1 2 **3** 4 5 6…

Rindex function: returns the position of the LAST occurrence of SUBSTR in STR. If POSITION is specified, returns the last occurrence at or before that position.

$[= 1;

$position = **rindex** ("aagaattcccgaattc", "gaattc");

print "Position = $position\n";

Yields the output = 11

Substring function: extracts a substring out of arg and returns it. To specify an end-point fo rthe extraction, the LENGTH parameter is added. To replace the extracted string with a flag, a final optional parameter can be added called the REPLACEMENT.

$seq = **substr(ARG, STARTPOS, LENGTH, REPLACEMENT);**

Lc function: returns a copy of the input string in lowercase letters

$out = **lc** ($dna);

Lcfirst(arg) function: returns the value of arg with the first character lowercased

Uc function: returns a copy of the input string in upper case

Ucfirst(arg) function: converts first character of string to upper case

length(arg) function: returns the length in characters of the value of the arg. If arg is omitted, returns length of $_.

Reverse(LIST) function: In array context, returns the LIST in reverse order. In scalar context, it returns the first element of LIST with bytes reversed

---