Perl – Lectures

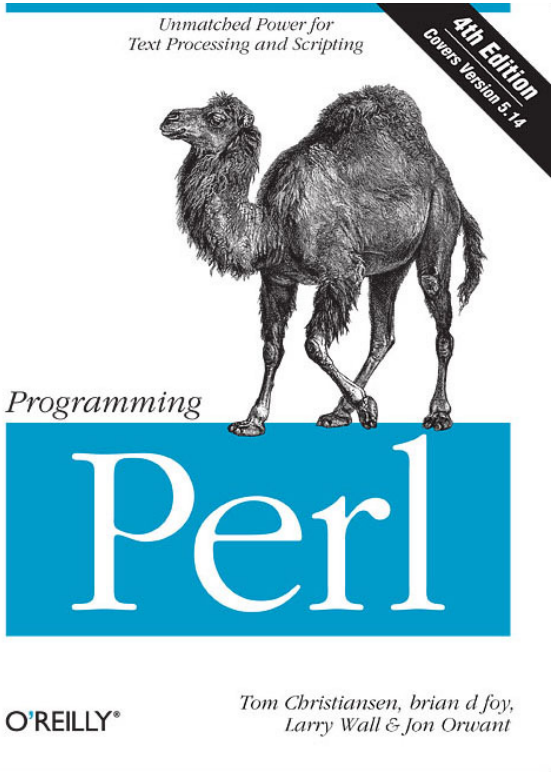
http://www.perl.org/

ILLINOIS INSTITUTE OF TECHNOLOGY

Jean-François Pombert, Ph.D.
Pritzker Science Center, rooms 296 (office) and 340 (Lab)

Optional - Books

Highly recommended



Programming Perl, 4th Edition

Print ISBN: 978-0-596-00492-7 | ISBN 10: 0-596-00492-3
Ebook ISBN: 978-1-4493-9890-3 | ISBN 10: 1-4493-9890-1

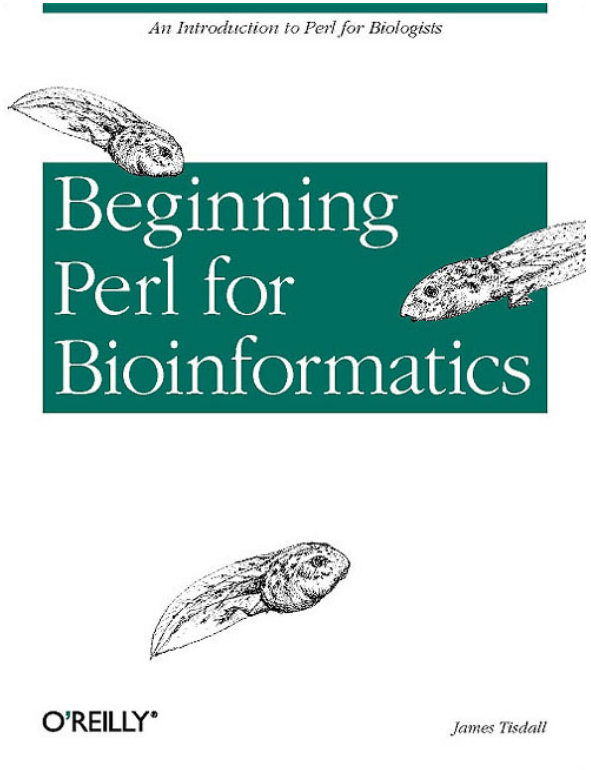
The Perl reference

Perl Programming for Biologists

ISBN: 978-0-471-43059-9

Highly liked by biologists

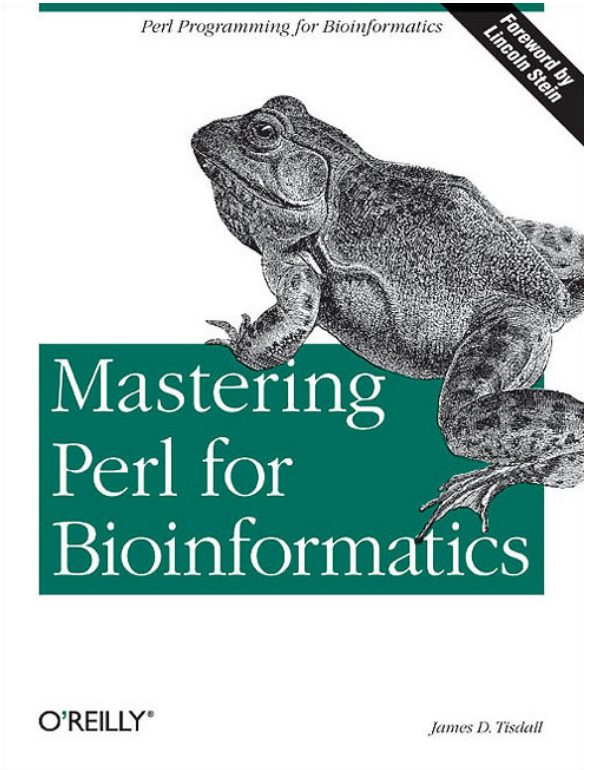
Not for everyone



Beginning Perl for Bioinformatics

ISBN: 9780596000806

Pseudocode heavy; wrong audience



Mastering Perl for Bioinformatics

ISBN: 9780596003074

More of the same

https://www.perl.org; free books

Getting help – Free resources

Perldoc (http://perldoc.perl.org/)

Perlmonks (http://www.perlmonks.org/)

Perl Maven (https://perlmaven.com/) ## One of my favorites

Looking for examples?

Many examples are available in online repositories, like GitHub or Sourceforge

A few examples from my lab:

https://github.com/PombertLab/3DFI

https://github.com/PombertLab/SSRG

What is Perl?

English-like programming language ## Larry Wall, 1987

Developed for text manipulation

- A 'glue language' for Unix
- One of the standard in bioinformatics ## Python is also very useful

Perl's motto? TIMTOWTDI - *There is more than one way to do it!*

Pre-installed on Linux & MacOSX (MS Windows; install ActivePerl or Strawberry Perl)

BioPerl modules ## Perl tools for bioinformatics, genomics and life science

Why Perl? – one example

Manual copy & paste

(@ 5 seconds/operation)

- 100,000 C&P = 500,000 sec
- 500,000/3600 sec/hour ? 139 hours
- 5 ½ days non-stop
- Made an error? Need to redo it? Ouch…

+ incredibly boring…

Writing a script

- 5 to 15 min to write?
- 5 to 60 sec to run?
- Scripting error? Easy to fix.
- Reusable code

+ rarely boring…

The Perl script

A human-readable text file	## Although the code can be obfuscated!
Often less than one page	## Shorter scripts are easier to debug/maintain
Does not require compiling	## The Perl interpreter will translate your code in machine language
Easy to modify (if properly written)	## A robust design is always good

You don’t need to know everything

It works? Good enough!

Perl is a good introduction to computer programming

Copy & Paste – Yes, you can

Code recycling is expected

As you get better, you won’t rely on C&P anymore

Give credit where it is due

Recommended text editors

All platforms

vscode https://code.visualstudio.com/
Atom https://atom.io/

Linux

SciTE http://www.scintilla.org/SciTE.html
Gedit https://projects.gnome.org/gedit/

MS Windows

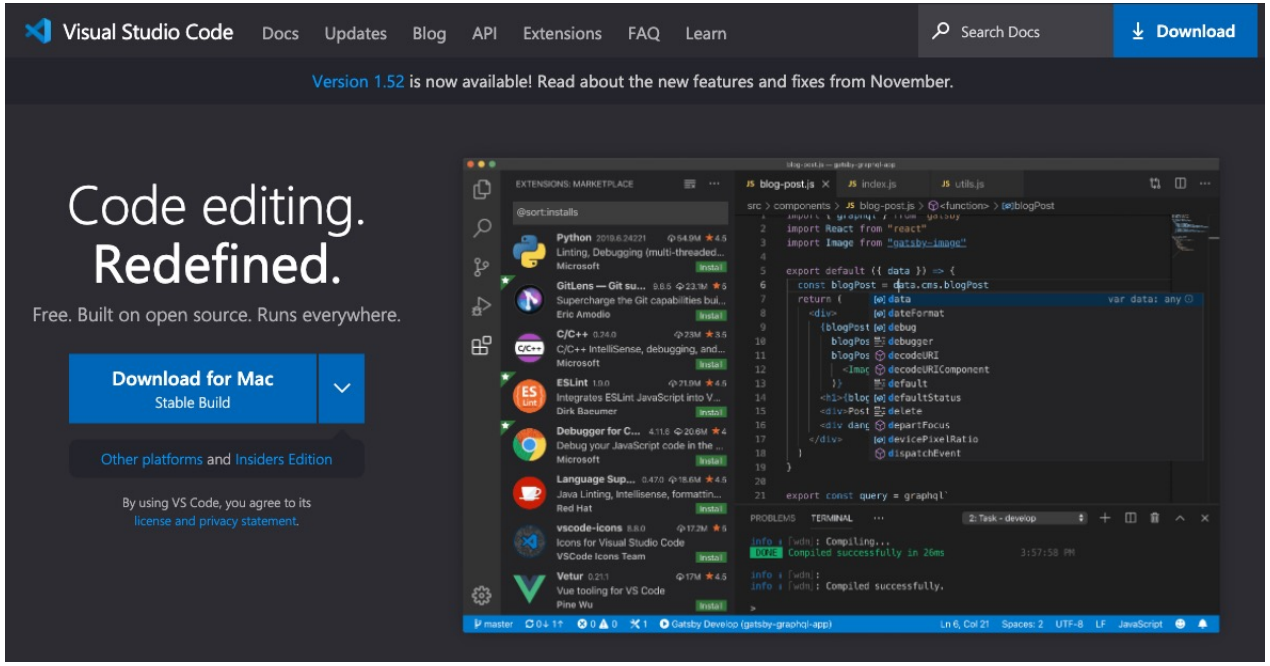
Notepad++ http://notepad-plus-plus.org/
SciTE https://www.scintilla.org/SciTE.html

MacOSX

Xcode https://developer.apple.com/xcode/
TextWrangler https://www.barebones.com/products/bbedit/

now part of BBEdit, can be used for free
but with fewer options

My favorite, highly recommended:



A good editor will colorize your code

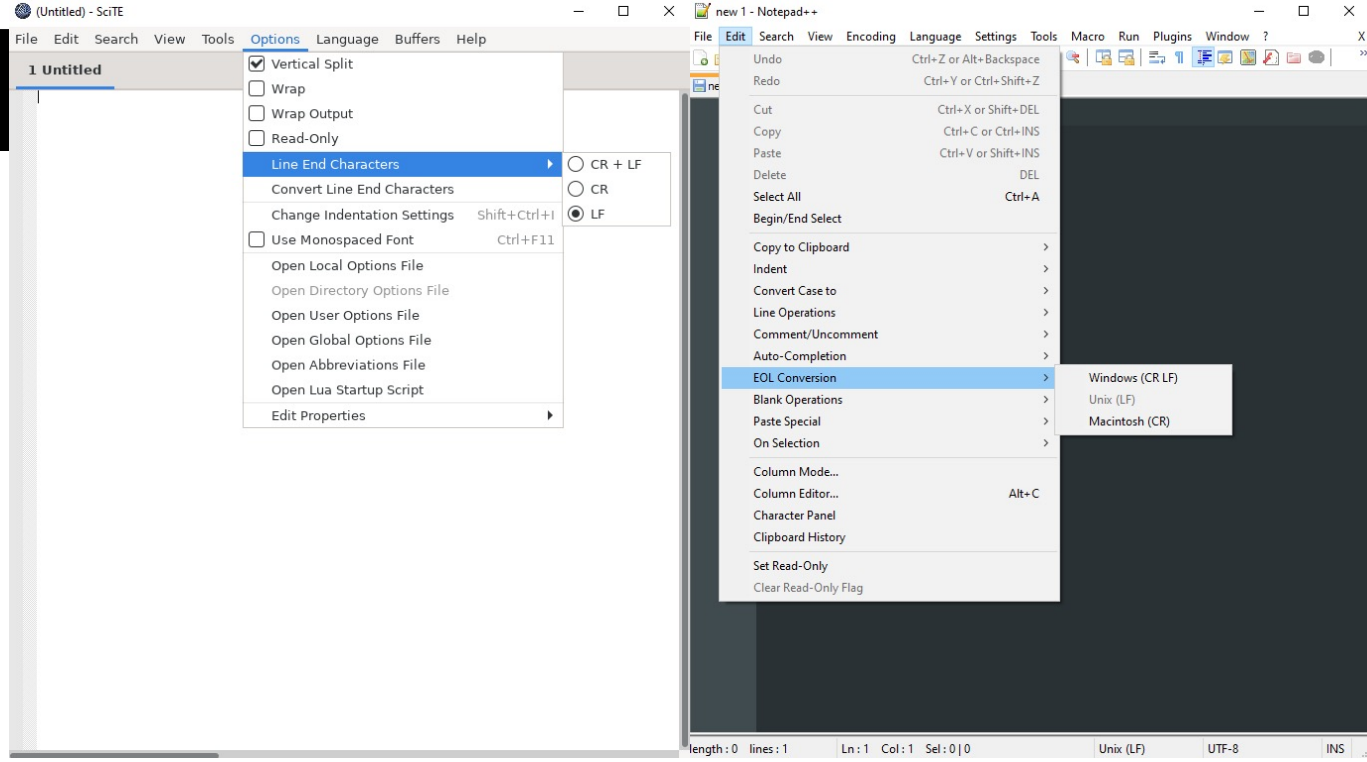
Line breaks – Unix/DOS/Mac

Line breaks (aka newlines, end of lines) differ between operating systems (OS)

Linux/MacOSX	LF	"\n"	## LF -> Line Feed
MS Windows/DOS	CR-LF	"\r\n"	## CR -> Carriage Return
MacOS ≤ 9	CR	"\r"	## Really old; unlikely nowadays

Wrong line breaks cause issues with many programs; good text editors will interconvert them

```
jpombert@DESKTOP-31J5PC0: /mnt/c/Users/jfpombert/Desktop/test
jpombert@DESKTOP-31J5PC0:/mnt/c/Users/jfpombert/Desktop/test$ dos2unix run_GESAMT.pl
dos2unix: converting file run_GESAMT.pl to Unix format...
jpombert@DESKTOP-31J5PC0:/mnt/c/Users/jfpombert/Desktop/test$
```



dos2unix

unix2dos <-> dos2mac <-> mac2unix

Available by default on most Linux machines ## Must be installed on Ubuntu

Interconverts between line formats ## Safe; only modify files in the right format

Alternatively, using text editors

SciTE Options -> Line End Characters

Notepad++ Edit -> EOL conversion -> Windows, Unix or Mac

end of line

Perl programming – In a nutshell

Variables	## scalars ($; single variables), arrays (@; list of variables), hashes (%; databases)
Filehandles	## *i.e.* shortcuts that 'handle' files
Operators	## e.*g.* math (addition, subtraction) and comparisons (ne, eq, >=)
Functions	## Bits of code used for specific task; *e.g.* print, open, system
Control structures	## Control flows; *e.g.* if/elsif/else statements, while and for loops
Regular expressions	## Equations used to find text patterns
Comments	## This is a comment; comments are delimited by hashtags (#)

The very common mistakes

Typos	## *e.g.* S instead of $; sequence instead of sequences
Missing a semi-colon (;)	## Statements must be closed
Unclosed quotes, { }, ( ), or [ ]	## Quotes, curly braces, parentheses, brackets must be closed too!
Wrong quotes ('$a' != "$a")	## Literal (') and interpolation (") quotes behave differently

Pseudocode your life

Plan your code before writing it out

- Do I need to read from a file?
- Must I iterate line by line through it?
- Do I need a regular expression to find patterns?
- Should I store variables? Into scalars, arrays, and/or hashes?

Script anatomy

Perl interpreter location

Pragmas/modules

Usage definition

Comment(s)

Loop

File handling

Loop-within-loop

File handling (optional)

If, elsif, else…

```
#!/usr/bin/perl
use strict;
use warnings;
my $usage = 'perl fasta_to_string.pl *.fsa';
die $usage unless @ARGV;
## convert the content of single fasta files to single & one-line DNA strings
while (my $fsa = shift @ARGV) {
    open IN, "<$fsa" or die "cannot open $fsa";
    $fsa =~ s/\.fsa$//;
    open OUT, ">$fsa.string";
    my @string = ();
    while (my $line = <IN>) {
        chomp $line;
        if ($line =~ /^>.*$/) {
            next;
        }
        else {
            push (@string, $line);
        }
        my $DNA = join('', @string);
        $DNA =~ s/\s//;
        print OUT "$DNA";
        @string = ();
    }
    close IN;
}
exit;
```

A very simple script

Script: print "This script works!\n";

To use:

```
perl -e 'print "This script works!\n"'
perl name_of_file_containing_the_script.pl
```

What will happen? This script works! will appear in the shell (it will be printed).

TIMTOWTDI (totally odd, but works nonetheless)

```
print 'This '.
'script '.
'works! '.
"\n";
```

BTW – By convention

Perl scripts end with .pl or .perl

Required? No, but makes a lot of sense

.py (python), .sh (shell), .r (R) ...

BASH, ZSH, TCSH...

– Comments

Useful annotations (disregarded by the Perl interpreter)

Not part of scripts *per se*

This is a comment ; # This one too!

TIP: Want to test the behavior of some code? Why not disable/enable it with comments?

#! – The shebang (hash + bang)

A very unique comment at the top of your script

Required to make scripts executable

Points the OS to the desired script interpreter (*e.g.* Perl, Python, Bash, Rscript):

```
#!/usr/bin/perl
#!/usr/bin/bash
#!/usr/bin/python
```

To run a Perl script

```
perl -e 'oneliners'          ## Using one liners; I don’t recommend it unless script is simple
perl script.pl               ## Invoking perl; does not require the script to be executable
chmod a+x script.pl; ./script.pl          ## Making your script executable, then running
                                          ## the script from its absolute or relative path
```

Print – Printing information

The function **print** displays information to the standard output ## Usually the shell

Print is not the same as **printf** or **sprintf** ## In the later two, f => format

Print is simple to use ## I use it a lot for debugging statements

Interpolation

Interpolation replaces a variable or a metacharacter by its content

''	literal quotes	## '$value' means exactly '$value'
""	Interpolation quotes	## "$value" means the content of $value

REMINDER: ’ and ” are not the same as ' and ". Non-programming text editors such as Microsoft Word will often corrupt these quotes.

\ – The great escape

The **backslash** (\) is also known as the **escape** character in Perl

It converts **meta** characters into **regular** ones and vice-versa

Remember regular expressions? Same thing.

$.$ – Concatenate strings

```
print "alma "."matter";                ## Works with print
print "alma matter";                   ## TIMTOWTDI
$DNA = 'ATCGCTT'.'TGTTTTAA';           ## Works with strings too
```

Perl – Variables (and sigils)

	$	Scalar	individual values
	@	Arrays	lists
sigils	%	Hashes	database-like ‘key => value’ lists
	&	Subroutines	reusable code chunks (intermediate to advanced Perl)
	*	Typeglobs	wildcard as in Bash shell, rarely used in Perl

Variables – SNAFUs*

Don’t use spaces

Don’t use brackets or parentheses (reserved for specific tasks)

Don’t start with a number ($1, $2, $3… => special Perl values VERY useful with Regex)

Don’t start with underscores ($_ and @_ are predefined Perl variables)

*S(ituation) N(ormal) A(ll) F(u**ed) U(p)

Readability is key

Use logical names that describe the values

Use underscores (_) to increase readability if needed

Longer names are hard to maintain

Describing names with comments (#) is often a good idea

Variables – Scalars ($)

Numbers ## integers (whole number) or floating points (*i.e.* with decimals)

Octal, hexadecimal, binary numbers supported (advanced Perl)

Strings

Assigning numeric literals ($)

$students = 25;	GOOD
$students = 25.01;	GOOD
$students = 25,000,000;	BAD
$students = 25_000_000;	GOOD

Commas are not allowed, they are used as list separators

Assigning string literals ($)

$students = 'present';	GOOD	
$students = '$present';	GOOD	***Different meanings***
$students = "$present";	GOOD	
$students = ‘$present’;	BAD	***Wrong quotes*** ‘ ’ ≠ ' '
$students = “$present”;	BAD	“ ” ≠ " "
$students = "$present"	BAD	***Missing semicolon***

Exercise 1 – Interpolation

1) Create a small Perl script that says in the shell:

Why do I have a "feeling" this won't work…

2) What happens?

3) Try with single quotes (')

4) Try with double quotes (")

Exercise 2 – Setting values

1) Create a Perl script that sets $first as 2 and $second as 7, then prints in the shell:

```
My first value is $first          ## $first => INTERPOLATED VALUE
My second value is $second        ## $second => INTERPOLATED VALUE
```

2) Modify the values of $first and $second to three and humongous, respectively. Run the script again.

https://perlmaven.com/perl-arrays
https://www.perlmonks.org/?node_id=739305

Arrays (@)

Store lists of values

Perl arrays are read from left to right ## But we can also write them from top to bottom

Perl arrays start at **zero**

Defining arrays (@)

```
@array = ();                                    # Empty
@array = ('1', 'gamma', 'Roger', 'Xanadu');     # Now contains 4 elements
@array = (1, gamma, Roger, Xanadu);             # TIMTOWTDI 1
or      $array[0] = '1';                        # TIMTOWTDI 2
        $array[1] = 'gamma';
        $array[2] = 'Roger';
        $array[3] = 'Xanadu';
```

1st element is ZERO!!!

Modifying arrays (@)

Adding to @

- push `push (@array, item);` ## Adds item at the end of the list (bottom or right)
- unshift `unshift (@array, item);` ## Adds item at the top of the list (top or left)

Removing from @

- shift `shift (@array);` ## Removes the 1st item from the list (top or left)
- pop `pop (@array);` ## Removes the last item from the list (bottom or right)

Sorting @

- sort `@array = sort(@array);` ## Sorts (right) then saves (left) @array alphabetically
- reverse `@array = reverse(@array);` ## Reverses (right) the list order then saves it (left)
- scalar `$number = scalar(@array);` ## Counts list size (right) and stores it in $number (left)

Modifying arrays (@) – By position

Top of @ (left side)

- shift `$variable = shift@array;` ## Removes top item from @array; stores it in $variable
- unshift `unshift (@array, $variable);` ## Adds $variable to the top of the @array

Bottom of @ (right side)

- push `push (@array, item);` ## Adds $variable to the bottom of the @array
- pop `$variable = pop@array;` ## Removes bottom item from @array; stores it in $variable

Exercise 3 – Create an array

1) Create a script containing an empty array named @testing
2) Add lettuce, tomatoes, cheese to @testing at the bottom (right) of the array
3) Make your script return in the shell:

```
My element $testing[0] value is: lettuce      ## interpolated value of $testing[0]
My element $testing[1] value is: tomatoes     ## interpolated value of $testing[1]
My element $testing[2] value is: cheese       ## interpolated value of $testing[2]
```

4) Put bacon at the start of @testing, reprint the above values

Exercise 4 – Modify an array

1) Use @testing from exercise 3
2) Add cucumbers, mayo, and red peppers at the top (left)
3) Sort the array and return all elements in one line
4) Sort the array and return all elements in one line in reversed order
5) Return a line that says: Number of items in array = $number
   ## where $number is the interpolated value

@ARGV – A very special array

Created automatically

Holds values from the command line (Arguments variables)

Incredibly powerful yet simple with globs

Exercise 5 – Understanding @ARGV

1) Create a script that will print the content of @ARGV
2) Reverse the order of @ARGV
3) Calculate its size, and make your script return this info
4) Type: perl yourscriptname.pl word1 word2 word3 word4 word5
5) What happens?
6) Try perl yourscriptname.pl *.txt, *.pl or *.perl

https://perlmaven.com/perl-hashes
https://www.learn-perl.org/en/Hashes

Hashes (%) – high speed indexes

Database-like key => value lists

Keys must be unique, values can be repeated

Out-of-order lists

Assigning hashes (%)

```
## Example: A database of famous hockey players and their unique nicknames

%hockey = (TheRocket, 'Maurice Richard', SuperJoe, 'Joe Sakic', MrHockey,
'Gordie Howe');

## TIMTOWTDI (much easier to read)

%hockey = (
        TheRocket => 'Maurice Richard',
        SuperJoe => 'Joe Sakic',
        MrHockey => 'Gordie Howe',
);
```

Adding keys + values to hashes (%)

```
%hockey;                                          ## Initializing the hash
$hockey{TheRocket} = "Maurice Richard";           ## Adding key + value to
                                                  ## the hash
```

key value

```
$hockey{SuperJoe} = "Joe Sakic";                  ## Same
$hockey{MrHockey} = "Gordie Howe";                ## Same

$hockey{$nickname} = "$real_name";                ## Using variables instead
```

Biology example 1 – Sequences

Hashes are great storage containers for sequences (DNA, RNA and/or proteins)

```
%sequences = (
   PCR01 => 'AAAGAAACGATAGAACGGTTATTCGGAACAGCTAAAGAATATCATAACTTGTACGGG',
   PCR02 => 'TTACTTTAGCGTGTCTAAATATCAAAAAATTGGTAAAAATGATGACAGGA',
   PCR03 => 'TTGCCAGTGAAAATTTCCAAAGTCATTCGTCAACCAGTTGGAACCAGCAGGGATTTACCCCCC',
   RNA01 => 'AGAGGGGGAUGGUAAGGAGAAUUGGGCUGUACAGUUGGUUCAAAUUUUC',
   PROT1 => 'MTIYKKKSIMLLSKKKDVLPLQEKQEAFVANCKTVLRRRLSGNKLIAYKERLSLTTIQKE',
);
```

Biology example 2

Universal codon table

We can translate genes into proteins with Perl

Note that the code on the right is much easier to read when colorized by a Perl-enabled text editor

```
%aa = (
    'tca'=>'S','tcc'=>'S','tcg'=>'S','tct'=>'S',
    'ttc'=>'F','ttt'=>'F',
    'tta'=>'L','ttg'=>'L',
    'cta'=>'L','ctc'=>'L','ctg'=>'L','ctt'=>'L',
    'tac'=>'Y','tat'=>'Y',
    'taa'=>'_','tag'=>'_','tga'=>'_',
    'tgc'=>'C','tgt'=>'C','tgg'=>'W',
    'cca'=>'P','ccc'=>'P','ccg'=>'P','cct'=>'P',
    'cac'=>'H','cat'=>'H','caa'=>'Q','cag'=>'Q',
    'cga'=>'R','cgc'=>'R','cgg'=>'R','cgt'=>'R',
    'ata'=>'I','atc'=>'I','att'=>'I','atg'=>'M',
    'aca'=>'T','acc'=>'T','acg'=>'T','act'=>'T',
    'aac'=>'N','aat'=>'N','aaa'=>'K','aag'=>'K',
    'agc'=>'S','agt'=>'S','aga'=>'R','agg'=>'R',
    'gta'=>'V','gtc'=>'V','gtg'=>'V','gtt'=>'V',
    'gca'=>'A','gcc'=>'A','gcg'=>'A','gct'=>'A',
    'gac'=>'D','gat'=>'D','gaa'=>'E','gag'=>'E',
    'gga'=>'G','ggc'=>'G','ggg'=>'G','ggt'=>'G'
); ## aa = amino acids
```

Hashes (%) – Cheat sheet

```
%hash = ();                                  ## Create an empty hash

$hash{key} = 'value';                        ## Add a single key to an existing hash

        ## ^ Notice the curly braces

@array = keys(%hash)                         ## Storing all keys from the database into an array

$num = scalar(keys(%hash));                  ## Count # of keys in hash; same as scalar(@array)

for (keys %hash){ print "$_ $hash{$_}\n"; }  ## Printing an hash with a for loop

for (keys %hash){ delete $hash{$_}; }        ## Empty hash

if (exists $hash{$key}){ do something; }     ## Verify if key is present (True or false)
```

pseudocode

For loops; we'll see them a bit later

If, elsif, else; we'll see those later too

Exercise 6 – Hashes (to hashes?)

Create a script that

1) Initializes an empty hash called %sequences
2) Adds gene01 (GTCGTAGTT), gene02 (TCATGATTT) and gene03 (TGATGGTC) to the hash in one single addition
3) Make your script print the sequence of gene01 in the shell
4) Add gene04 (ATGCGTAATC) to the hash
5) Print the full hash in the shell
6) What happens?

The diamond operator (<>) – I/Os

<	Input	## Same operators as for the Bash shell
>	Output (write/overwrite)	
>>	Output (appends)	

Perl – Filehandles

Filehandles are shortcuts to files ## *i.e.* they handle files!

Used to read from/write to files

```
open INPUT, "<", $file;
open $input, "<", $file;
```

We could also use variables instead of
capital letters

https://perlmaven.com/stdout-stderr-and-redirection

STDIN/STDOUT – Default I/Os

STDIN → A filehandle ## Standard input

STDOUT ## Standard output

In Linux, the default (standard) input/output (I/O) are from/to the shell

CAPITAL letters used for filehandles are not mandatory, but easier to read

https://perlmaven.com/read-from-stdin

STDIN/STDOUT – How to?

```
$value = <STDIN>;
chomp ($value = <STDIN>);
print STDOUT "something"
```

```
$value = <>;                ##TIMTOWTDIs
chomp ($value = <>);        ##TIMTOWTDIs
print "something"          ##TIMTOWTDIs
```

chomp – Removing newlines

```
chomp $value;  ## Removes newline character (\n) from the end of the string
```

Newlines can create problems with your code

Removing them by default is usually a good idea

https://perlmaven.com/open-and-read-from-files

Reading files (<) – TIMTOWTDI

```
open IN, '<name_of_file.txt';      ## Filehandle name is arbitrary, can be anything (e.g. open FASTA)
open IN, "<$file";                 ## Short
open IN, "<", $file;               ## A bit more verbose
open (IN, "<", $file);             ## Parentheses are optional
```

The filehandle IN (for input) is commonly used in Perl

Loading files (into arrays)

```
@array = <FILEHANDLE>;                                  ## Lines ARE NOT chomped
use File::Slurp; @array = read_file ("$file", chomp => 1);     ## using Perl modules
use Tie::File; tie @array, 'Tie::File', "$file";     ## using Perl modules ; NOT chomped
```

https://perlmaven.com/writing-to-files-with-perl

Writing to files (>) – TIMTOWTDI

```
open OUT, '>name_of_file.out';        ## OUT is arbitrary too, can be anything
open OUT, ">$file";                   ## Parentheses are optional; open (OUT, ">$file");
open OUT, ">", $file;
print OUT "desired output";           ## print to file instead of STDOUT
```

Exercise 7 – Reading from files

Create a script that:

1) Opens (reads from) the file Dracula.txt
2) Fills an array called @Bram with the file (for testing purposes)
3) Prints the array in the shell (see if it worked)
4) Make the script open the file via command line instead with @ARGV
5) Print it

Exercise 8 – Writing to files

Create a script that:

1) Opens (read from) the file Dracula.txt
2) Fills an array called @Bram with the file
3) Saves it (writes) to Classics.txt (less Classics.txt to see if it worked)
4) Make the script open and save the file via command line instead with @ARGV
5) Test it

Exercise 9 – Multiple I/Os

Create a script that:

1) Can open two text files from the command line input
2) Displays these two files back-to-back
3) Concatenate these two files as one, and prints the concatenation to 2 different outputs (nameoffile1.out and nameoffile2.out)

Exercise 10 – Understanding chomp

Create a script that:

1) Takes an input from STDIN
2) Prints: Here is the complete value of STDIN: $value (use \n) ## interpolated $value
3) Chomps the STDIN value
4) Prints: Here is the chomped value of STDIN: $value (use \n) ## interpolated $value
5) Notice the difference?

Arithmetic operators – Numerals

Ye Olde Math (by default)

Floating-point: 5/2 = 2.5 ## FLOPS = FLoating-point Operations Per Second

Integer: int(5/2) = 2 or int(5-2.5) = 2 ## Fractions are discarded with integers

Modulo: 12 % 3 ## Returns the remainder of a division; great with codons to identify frames!

Assignment operators – Math

Apply the math on the right to the value on the left

$value = 10; (we are setting the numerical value to 10)

$value += 10; (we are increasing the numerical value by10)

Not the same, unless initial value was zero

Any math operator followed by = ## += -= *= /=

Careful about multiplications

The asterisk (*) is the multiplication symbol

The (x) symbol may surprise you ## x is the repetition symbol

Exercise 11 – (*) *vs.* (x) multiplications

Create a script that:

1) Sets a first value as the result of 2*5
2) Sets the second value as the result of 2x5
3) Prints each value on a line with a short statement describing what you did
4) Notice the difference?

Auto-increment/decrement

$value++; $value--; Modify after returning value

++$value; --$value; Modify before returning value

Exercise 12 – Math and increments

Create a script that:

1) Takes many inputs from @ARGV
2) Returns the number of inputs from @ARGV that you typed in
3) Says: Now I'm multiplying by 10: $value ## interpolated
4) Says: Here is the auto-incremented value: $value ## interpolated
5) Says: That squared increment is big (or not): $value ## interpolated

.= – Concatenate scalars

```
$sequence = 'ATGCCA'.'TTTTAA';        ## We can concatenate with a dot
$sequence = 'ATGCCA';                 ## TIMTOWTDI with .=
$sequence .= 'TTTTAA';                ## Adding to the end (right, 3') of $sequence
print "Sequence = $sequence\n";       ## Sequence = ATGCCATTTTAA
```

very useful when working with fasta files!

Exercise 13 – Concatenate sequences

Create a script that:

1) Asks for a DNA sequence from STDIN, then sets it as a value
2) Asks for a second DNA sequence from STDIN, then sets it as a value
3) Asks for a third DNA sequence from STDIN, then sets it as a value
4) Prints on the same line the concatenated sequences

Comparison operators

>= ne gt

Operators differ for numbers and words

Using the wrong type will make your scripts behave oddly

Comparison operators – Numbers

<	Smaller (lower) than
>	Bigger (greater) than
<=	Smaller or equal to
>=	Bigger or equal to
==	equal to
!=	not equal to

Math symbols are used as comparison operators for numbers

Comparison operators – Words

lt	lower (smaller) than
gt	greater than
le	lower or equal to
ge	greater or equal to
eq	equal to
ne	not equal to

The important ones

Letters are used as comparison operators for words

Perl – Control structures

```
function ( condition(s) to be evaluated ) { code to be executed }   ## Pseudocode
```

Block statement

```
## examples
if ( 256 > 128 ) { print "256 is bigger than 128, obviously\n"; }   ## If; conditional statement
for (0 .. 10) { print "My number is $_\n"; }   ## For; loop
```

https://perlmaven.com/if

if, elsif, else – Truth *vs.* untruth

Testing for simple and/or multiple conditions

Fantastic with regular expressions, m// and =~

```
if ( condition is true ) { do this; }          ## Pseudocode
unless ( condition is true ) { do this; }      ## Unless is the opposite of if
```

Exercise 14 – Computer hardware trivia

Create a script that:

1) Asks the question: What is the total number of CPU cores on this workstation?
2) Takes your answer from the shell
3) Validates your answer against the correct answer (*e.g.* **32**)
4) Returns: Correct! (in the shell if answer is true)
5) Returns: Wrong! The correct answer is **32** ## or the number you have set

if, elsif, else – Multiple conditions

```
&&      and     ## Both must be true
||      or      ## At least one must be true
if ( ($a < $b) && ($c eq 'something') ) { pseudocode; }     ## TIMTOWTDI
if ( ($a < $b) and ($c eq 'something') ) { pseudocode; }    ## TIMTOWTDI
```

if, elsif, else – Nested conditions

```
if ( $a < $b) {
        if ($c eq 'something')  { pseudocode; }
        elsif ($c ne 'something') { pseudocode; }
}
```

Exercise 15 – A tougher quiz!

Create a script that:

1) Asks the questions: What is the total number of CPU cores on this workstation? and What is the amount of RAM (in Gb) available?
2) Takes your inputs, then validate them against the correct answers (*e.g.* **32** and **256**)
3) Returns: Correct in both cases! (if both answers are true)
4) Returns: The correct amount of RAM is **256** Gb if RAM input is wrong
5) Returns: The correct number of CPU cores is **32** if CPU cores input is wrong
6) Returns: Correct values are: CPU cores=32, RAM=**256** Gb if both inputs are wrong

You can of course change the values in red to whatever you want

Loops

Repeat structures

Run until conditions/expressions are met

```
for (@ARGV) { pseudocode; }                    ## For loops
while ($file = shift @ARGV) { pseudocode; }    ## While/until loops
```

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

Potential RAM hog!!!

See StackOverflow for a great explanation of the memory issue

Foreach/For – BASH-like loops

```
foreach and for are synonyms                    ## Since Perl 2.0
Work great with arrays (@)                      ## Fine with small/medium-sized arrays or files
foreach (@array) {print "$_\n";}                ## Prints the content of an array
for (0..$#array) {print "$array[$_]\n";}        ## TIMTOWTDI; $#array means end of the array; $_  is the number in the loop
for $i (0..$#array) {print "$array[$i]\n";}     ## TIMTOWTDI; resembles bash loops
```

until the end of the array

https://perlmaven.com/for-loop-in-perl

For – C++ style loops

```
for ($i = 0; $i < $n; $i++){ pseudocode; }
```

Starting value

Condition evaluated

increment

Runs pseudocode as long as $i is smaller than $n

Same style as C/C++

Great for sliding windows

Substr – Creating substrings

```
$mRNA = "AUGCCUGCGUUAGCGUUAUAA";
$codon = substr($mRNA, 0, 3);          ## Like arrays, the first element of a string is zero
```

start step

```
$stop_codon = substr($mRNA, -3, 3);    ## You can start from the end of a string by using a negative number
```

https://perlmaven.com/while-loop

While (until) – Memory friendly

```
while ( condition is true ) { run this code; }      ## Stops when condition becomes false
until ( condition is true ) { run this code; }      ## Stops when condition becomes true
while ( $x = shift @array ) { print "$x\n"; }       ## Iterate through all scalars in @array
while ( $line = <INPUT> ) { print "$line\n"; }      ## Iterate through all lines in INPUT
```

Memory friendly + I/O efficient

While – Iterating through an array

```
@array = (
        'item1',
        'item2',
        'item3'
);
while ($x = shift @array) {
        print "Removing item $x from \@array\n";
        $num = scalar@array;
        print "Number of items remaining in \@array = $num\n";
}
```

While – Reading a single file, line per line

```
open IN, "<", "$file";
$count = 0;   ## Initializing a counter; we will autoincrement it below
while ($line = <IN>) {
        chomp $line;
        print "$line\n";
        $count++;
}
print "My file had a total of $count lines\n";
```

A while of while! – Reading multiple files, line per line

```
while ($file = shift @ARGV){
        open IN, "<", "$file";
        open HEAD, ">", "$file.head";
        open RAW, ">", "$file.raw";
        while ($line = <IN>) {
                chomp $line;
                if ($line =~ /^>(\S+)/) { print HEAD "My fasta header  =  $1\n"; }
                else { print RAW "My fasta header  =  $1\n"; }
        }
}
```

One of the most useful Perl structure to understand/remember

=~ – The binding operator

Binds a scalar to a **pattern match** (works with m//, s///, tr/// [y///])

```
=~      ## if true, binds the string on the left to the operation the right
!~      ## if false, binds the string on the left to the operation the right
## The eq comparison operator does not work with regular expressions
```

The m// operator – Match

Matching words and/or regular expressions

$line =~ m/regular expression/

m is optional ## m/regular expression/ and /regular expression/ are TIMTOWTDIs

Exercise 16 – A typical while + m// script

Create a script that:

1) Opens the text file Ex16.txt
2) Runs through each line with a while loop
3) Prints all lines matching Hsw to a file named Hsw.out
4) Prints all lines matching contig to a file named contig.out
5) Prints all other lines to a file named Misc.out

The m// operator – Parentheses

Parentheses capture values/string for Perl to remember

m/(word1)abc(word2)/

$1 = word1, $2 = word2

Exercise 17 – Printing only what you want ()

Create a script that:

1) Opens the text file Ex17.txt
2) Runs through each line with a while loop
3) Match only lines with: FT /gene="$gene_name"
4) Returns only gene and $gene_name separated by a tab from these lines in Genes.out

e.g. ccsA

The s/// operator – Substitution

Substitution operator based on match ## Remember sed?

s/pattern_to_look_for/replaced_by_this_pattern/

Optional switches; g -> global, i -> case insensitive (s///gi)

The tr/// operator – Transliteration

tr/// and y/// are equivalent

tr/individual_characters_to_replace/replaced_by_these_ind_char/

Useful with DNA/RNA sequences

Working on RNA and DNA

Converting DNA to RNA

```
$dna = 'ATGCTCCA';        ## Your input DNA
$rna = $dna;              ## Copy content of DNA to RNA
$rna =~ tr/Tt/Uu/;        ## modify $rna (converting thymine to uracil)
```

Reverse complement a DNA sequence

```
$dna = 'ATGCTCCA';                ## Your input DNA, 5’ to 3’
$revc = reverse($dna);            ## Your DNA, now 3’ to 5’
$revc =~ tr/ATGCatgc/TACGtacg/;   ## Now complemented
```

Exercise 18 – DNA manipulation script

Create a script that:

1) Asks for a DNA sequence in STDIN
2) Asks: Type 1 if you want to convert it to RNA or type 2 to reverse complement it
3) Takes your input
4) Performs the desired task and returns it

System – Calling the operating system

```
system "cat $file";
```

Perl can run operating system programs

Works great on UNIX/Linux-like OS

An easy to way to automate repetitive commands

Fantastic with @ARGV

Exercise 19 – The Perl system operator

Create a script that:

Takes multiples fasta files as inputs

1) Prints: Aligning INPUT with clustalo… ## clustalo => sequence alignments
2) Runs the system command on each of these file:
   /opt/clustal/clustalo -i $file -o $file.out
3) Prints: Job done! in the shell when completed

https://perldoc.perl.org/functions/die.html
https://perlmaven.com/die

Do or die – Can’t do? Stop!

```
open OUT, ">", "$file" or die;                          ## Stops if file cannot be created
open OUT, ">", "$file" or die "Can’t write file!\n";    ## Stops + prints the statement to STDERR
$usage = "command line description";                    ## Stops if @ARGV is empty and prints the message
die $usage unless @ARGV;                                ## described in $usage to STDERR
```

Exercise 20 – Understanding do or die

1) Create a script that:
   1) Creates a filehandle INPUT reading from the file $ARGV[0]
   2) Dies if the file cannot be read.
2) Run the script without anything in @ARGV. What happens? Not very informative right?
3) Let's add the print statement "Cannot read file named $ARGV[0]\n" to the die condition.
4) Run the script again. More informative? Try it on an existing file. What happens?

Optional

Additional slides for the curious

https://perlmaven.com/here-documents

Perl here-document (aka string blocks)

Multiline strings definition

```
$string = <<"BLOCK";          ## Initiates the string, replace BLOCK
Enter your desired multiline string here   ## by name of your choosing
BLOCK                         ## Ends the string with BLOCK
```

```
my $options = <<'END_OPTIONS';
OPTIONS:

-h (--help)     Display this list of options

## Genetic distances
-mh     Evaluate genetic distances using Mash (Ondov et al. DOI: 10.1186/s13059-016-0997-x)
-out    Output file name [default: Mash.txt]
-sort   Sort Mash output by decreasing order of similarity

## Mapping options
-fa (--fasta)   Reference genome(s) in fasta file
-fq (--fastq)   Fastq reads to be mapped against reference(s)
-mapper         Read mapping tool: bwa, bowtie2 or hisat2 [default: bowtie2]
-caller         Variant caller: varscan2, bcftools or freebayes [default: varscan2]
-algo           BWA mapping algorithm:  bwasw, mem, samse [default: bwasw]
-threads        Number of processing threads [default: 16]
-bam            Keeps BAM files generated
-sam            Keeps SAM files generated; SAM files can be quite large

## VarScan2 parameters (see http://dkoboldt.github.io/varscan/using-varscan.html)
-indel                          Calculates indels       ## Runs mpileup2indel
-mc (--min-coverage)            [default: 15]           ## Minimum read depth at a position to make a call
-mr (--min-reads2)              [default: 5]            ## Minimum supporting reads at a position to call variants
-maq (--min-avg-qual)           [default: 28]           ## Minimum base quality at a position to count a read
-mvf (--min-var-freq)           [default: 0.2]          ## Minimum variant allele frequency threshold
-mhom (--min-freq-for-hom)      [default: 0.75]         ## Minimum frequency to call homozygote
-pv (--p-value)                 [default: 1e-02]        ## P-value threshold for calling variants
-sf (--strand-filter)           [default: 0]            ## 0 or 1; 1 ignores variants with >90% support on one strand

## FreeBayes/BCFtools (see https://github.com/ekg/freebayes/; https://samtools.github.io/bcftools/bcftools.html)
-ploidy                 [default: 1]            ## Change ploidy (if needed)

END_OPTIONS
```

Example of options defined using a here-document

https://github.com/PombertLab/SSRG/blob/master/get_SNPs.pl

Perl programming good practices

Annotate your scripts with useful comments

Use indentation

Use strict and warning pragmas

Enforce scoped variables (my)

*E:\Perl_scripts\GeneAlignment_SNPs_v2.pl - Notepad++

File Edit Search View Encoding Language Settings Macro Run Plugins Window ?

run_macse.pl | Codon_SNPs.pl | sort_soapSNP_v2.pl | GeneAlignment_SNPs_v2.pl

```perl
#!/usr/bin/perl
## Loads a list of ECUs from a text file, one ECU per line, then each corresponding ECI, ECII and ECIII sequence is loade
## Each aligned position is compared between the three genes, and written to the appropriate output files.
## This is possible because the genes have been aligned. A single frame shift will kill this script.
## Note 1: that if alignments are not based on codons, each gap containing gene will need to be verified.
## Note 2: The sequence must be on one line without a fasta header. Use fasta_to_string.pl script to convert to strings.
use strict;
use warnings;
my $usage = 'perl GeneAlignment_SNPs.pl list.txt';
die $usage unless @ARGV;
while (my $txt = shift @ARGV) {
    open IN, "<$txt" or die "cannot open $txt";
        while (my $line = <IN>) {
            chomp $line;
            if ($line =~ /^(ECU\d+.*)/) {
                my $ECU = $1;
                my $seq1= "ECI_$ECU.string";    ## match the file extention to that of the input file used
                my $seq2 = "ECII_$ECU.string";  ## match the file extention to that of the input file used
                my $seq3 = "ECIII_$ECU.string"; ## match the file extention to that of the input file used
                open (IN1, $seq1);
                open (IN2, $seq2);
                open (IN3, $seq3);
                open INVAR, ">$ECU.invar";
                open SNPs, ">$ECU.snps";
                open GAPs, ">$ECU.gaps";
                my $strain1 = <IN1>;
                my $strain2 = <IN2>;
                my $strain3 = <IN3>;
                my @ECI = split ('', $strain1);
                my @ECII = split ('', $strain2);
                my @ECIII = split ('', $strain3);
                my $length = scalar @ECI;
                my $end = $length-1;
                my $position = 1;
                foreach (0..$end) {
                    if (($ECI[$_] eq '-') || ($ECII[$_] eq '-') || ($ECIII[$_] eq '-')) {
                    print GAPs "$position\t$ECI[$_]\t$ECII[$_]\t$ECIII[$_]\n";
                    $position++;
                    }elsif (($ECI[$_] eq $ECII[$_]) && ($ECI[$_] eq $ECIII[$_])) {
                    print INVAR "$position\t$ECI[$_]\t$ECII[$_]\t$ECIII[$_]\n";
                    $position++;
                    }else {
                    print SNPs "$position\t$ECI[$_]\t$ECII[$_]\t$ECIII[$_]\n";
                    $position++;
                }
            }
        }
    close IN1;
    close IN2;
    close IN3;
    close INVAR;
    close SNPs;
    close GAPs;
    }
}
exit;
```

Perl source file

Most pragmas are lexically scoped

```
## We put them at the top so that they apply to
## everything afterwards
```

```
#!/usr/bin/perl
$x = 2;
print "$x\n";
use strict;
```

vs.

```
#!/usr/bin/perl
use strict;
$x = 2;
print "$x\n";
```

Pragmas

Modules influencing Perl behavior

https://perldoc.perl.org/index-pragmas.html ## List of available pragmas

strict	restricts unsafe constructs	## Prevents most SNAFUs
warnings	controls optional warnings	## Very useful debugger
threads	Perl interpreter-based threads	## For multithreading

https://perl.plover.com/FAQs/Namespaces.html

Perl – Package *vs.* lexical variables

Package variables	## Tied to a namespace; oldest variables in Perl
	## We have been using them by default
Lexical (private) variables	## Restricted to the designed lexical block(s)
	## Not tied to namespace

https://stackoverflow.com/questions/14190483/what-is-difference-between-namespace-package-and-module-in-perl
https://perlmaven.com/packages-modules-and-namespace-in-perl

```
#!/usr/bin/perl

package main; ## Default namespace
$x = 2;
print "main x = $x\n";
print "main x = $main::x\n";

package alt; ## New namespace called alt
$x = 5;
print "alt x = $x\n";
print "alt x = $alt::x\n";

package main; ## Reverting to default namespace
print "Switching to main: x is now $x\n";
```

Perl – Namespace, package and modules

Namespaces	## Containers of identifiers (*e.g.* variables, functions)
Package	## Switch between namespaces; defaults to package main
Modules	## Pre-written packages; contain useful code and subroutines (.pm)
	## Have their own namespaces

https://perlmaven.com/scope-of-variables-in-perl

```
#!/usr/bin/perl
use strict; use warnings;

$main::x = 2; ## This is how we define a package variable under the strict pragma
print "main x is: $main::x\n";

my $x = 5;  ## The lexical variable x does not clash with the package variable x
print "my x is: $x\n";
```

```
#!/usr/bin/perl
use strict; use warnings;

package main;
my $x = 5;
package alt;

print "my x is: $x\n";

## my is not tied to a namespace
```

Perl – Lexical variables

```
my      ## Most common lexical variable; my $scalar, my @array, my %hash
state   ## Declares a static lexical variable; requires the use 5.010; pragma
our     ## Shared between namespaces; lexical shortcut to package variable
        ## Can create problems, often better to use my or state
```

Perl – How to use my? Declare it on 1st use…

```
#!/usr/bin/perl
use strict; use warnings;
open IN, "<", "$ARGV[0]" or die "Can't read file: $ARGV[0]\n";
my @genes; my %products;
while (my $line = <IN>){
	chomp $line;
	if ($line =~ /^(\S+)\t(\S+)/){
		my $gene = $1;
		my $product = $2;
		push(@genes, $gene);
		$products{$gene} = $product;
	}
}
```

https://perlmaven.com/static-and-state-variables-in-perl

Perl – My *vs.* state

My – reinitialized every time

```
#!/usr/bin/perl
use 5.010;
for (1..10){
    my $x = 0;
    $x++;
    print "$x\n";
}
```

State – never reinitialized

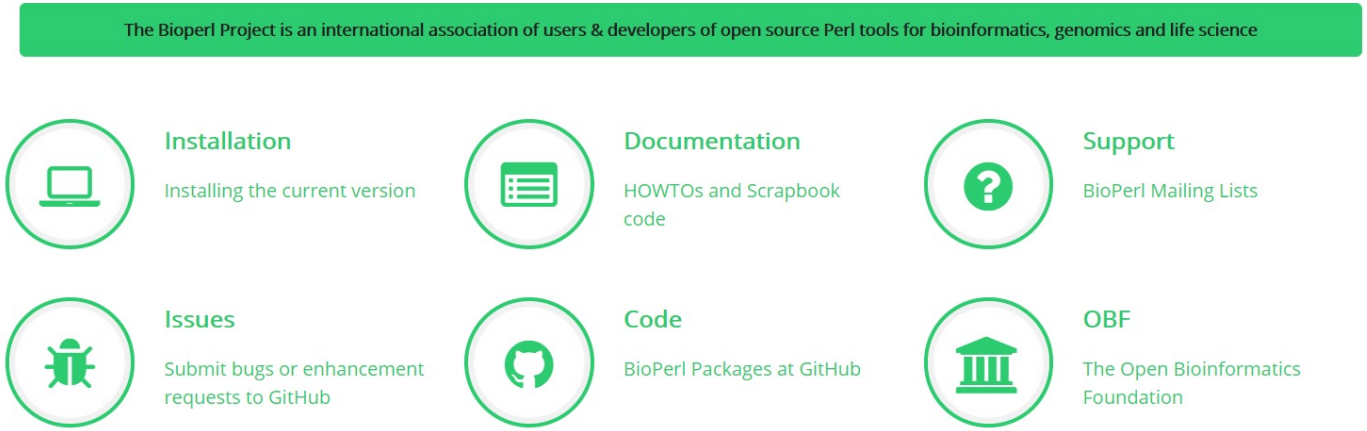
```
#!/usr/bin/perl
use 5.010;
for (1..10){
    state $x = 0;
    $x++;
    print "$x\n";
}
```

state requires the use 5.010; pragma

http://perldoc.perl.org/perlmod.html

"A module is a set of related functions in a library file"

Perl documentation



https://bioperl.org/ ## Perl tools for bioinformatics

https://metacpan.org/release/BioPerl ## BioPerl release on CPAN

Perl – Modules

Pre-written code for you to use ## Most are in Perl, some are written in C

Perl modules end with .pm ## May require dependencies

sudo cpanm Bio::Perl ## Install BioPerl with admin privileges

export PERL5LIB= $PERL5LIB: /path/to/modules ## To add Perl modules manually

http://perldoc.perl.org/Getopt/Long.html

```
use Getopt::Long qw(GetOptions);
```

qw = quote words
Imports the function named GetOptions
from the Getopt::Long module
into the current namespace

Command line switches

```
use Getopt::Long qw(GetOptions);    ## Parses the command line from @ARGV
```

Very useful

With this, commands can be out-or-order

e.g. get_SNPs.pl --fasta *.fasta --fastq *.fastq --mapper bowtie2 --caller freebayes --threads 16

e.g. get_SNPs.pl --threads 16 --fastq *.fastq --mapper bowtie2 --fasta *.fasta --caller freebayes

same thing

http://perldoc.perl.org/Getopt/Long.html
https://perldoc.perl.org/perlref.html

```
#!/usr/bin/perl
use strict; use warnings; use Getopt::Long qw(GetOptions);

my $usage = <<"END_OPTIONS";  ## Defining how to use the script in a multiline string
COMMAND perl.pl -fa *.fasta -out table.tsv
END_OPTIONS
die "\n$usage\n" unless @ARGV;

## Initializing variables
my @fasta;
my $output;

## Parsing @ARGV with GetOptions
GetOptions(
	'fa=s@{1,}' => \@fasta,
	'out=s' => \$output
);
```

GetOptions() – Basic usage

```
'flag' => \$flag                ## No value expected; on/off flag
'option=s' => \$scalar          ## Value expected is a scalar
'option=i' => \$integer         ## Value expected is an integer
'option=s@{1,}' => \@scalars    ## Expects at least one scalar
'option=i@{1,}' => \@integers   ## Expects at least one integer
```

An example - bam2fastq.pl

```
6   use strict; use warnings; use Getopt::Long qw(GetOptions);
7
8   my $usage = <<"OPTIONS";
9   NAME            $name
10  VERSION         $version
11  SYNOPSIS        Extract sequencing reads in FASTQ format from BAM alignment files
12  REQUIREMENTS    Samtools 1.3.1+
13
14  USAGE           bam2fastq.pl -b file.bam -t pe -e map -p reads -s fastq
15
16  OPTIONS:
17  -b (--bam)      BAM alignment file
18  -t (--type)     Alignment type: pe (paired ends) or single [Default: pe]
19  -e (--extract)  Reads to extract: map, unmap [Default: map]
20  -p (--prefix)   Output file(s) prefix [Default: reads]
21  -s (--suffix)   Output file(s) suffix [Default: fastq]
22  OPTIONS
23  die "$usage\n" unless @ARGV;
24
25  my $bam;
26  my $type = 'pe';
27  my $extract = 'map';
28  my $prefix = 'reads';
29  my $suffix = 'fastq';
30  GetOptions(
31          'b|bam=s' => \$bam,
32          't|type=s'      => \$type,
33          'e|extract=s' => \$extract,
34          'p|prefix=s' => \$prefix,
35          's|suffix=s' => \$suffix
36  );
37
38  ## Program + option check
39  my $samtools = `command -v samtools`; chomp $samtools; if ($samtools eq ''){print "\nERROR: Cannot find Samtools. Please install Samtools i
40  unless (($type eq 'pe') || ($type eq 'se')) {die "\nUnrecognized type $type. Please use 'pe' for paired-ends or 'se' for single ends\n";}
41  unless (($extract eq 'map') || ($extract eq 'unmap')) {
42          die "\nUnrecognized reads to extact: $extract. Please enter 'map' or 'unmap' to extract reads that map or do not map to the referen
43  }
```

Step 0 – Prepare a useful HOWTO (optional but always useful)

Step 1 - Declare your variables ($strings, @arrays or %hashes)

Step 2 - Declare your command line switches

b or bam
ft or type
Both will be accepted from the command line

**Step 3 - Reference the parsed command lines to the desired variables with **

Other examples:
SSRG.pl
get_SNPs.pl
sort_stats.pl

Exercise 21 – GetOptions() and CMD lines

Create a script that:

1) Takes 1 or more fasta file from the CMD (e.g. -fasta *.fasta)
2) Takes a specific output name from the CMD (e.g. -out mash.txt)
3) Takes a flag option to sort the output (e.g. -sort)
4) Runs the sketch segment of MASH: ## use system
   mash sketch @fasta -o reference.msh ## @fasta = *.fasta values from 1)
5) Runs the dist segment of MASH:
   mash dist reference.msh @fasta > $out
6) If sort is entered, executes the following block: { system "echo Sorting out Mash results";
   system "sort -gk3 $out > $out.sorted"; }

MASH is a tool that estimates genetic distances

Exercise 22 – Defining those GetOptions()

Edit your script from Exercise 21 to explain its usage.

1) Define a short usage example with $usage.
2) Explain thoroughly the usage with a Perl here-document like 'OPTIONS' to capture your usage definitions
3) Edit your script so that it will display the short $usage message if nothing is entered in @ARGV
4) Edit your script so that it will display the 'OPTIONS' block if the -h or -help option is entered in @ARGV

https://perlmaven.com/qx

Backticks – `command`

Executes commands, like system

Captures the STDOUT, very useful!

```
my $passwd = `makepasswd -l 12`;
chomp($passwd);
```

Here the output of makepasswd will be fed to $passwd

Yes, the values captured by backticks include the \n characters

```
#!/usr/bin/perl
use strict; use warnings; use Getopt::Long qw(GetOptions);

my $usage =<<"OPTIONS"; ## Defining options
SYNOPSIS      Generate passwords with makepasswd
COMMAND       bacticks.pl -l 12 -n 10 -o output.txt

-l     length of passwords
-n     number of passwords
-o     output file
OPTIONS
die "\n$usage\n\n" unless @ARGV;

my $len; my $num; my $out;
GetOptions( ## Creating CMD line switches
      'l=i' => \$len,
      'n=i' => \$num,
      'o=s' => \$out
);

open OUT, ">", "$out"; ## Creating passwords
for (1..$num){
      my $passwd = `makepasswd -l $len`;
      chomp $passwd;
      print OUT "$passwd\n";
}
```

Usage definition

Creating command lines switches with GetOptions();

Creating passwords with backticks

https://perldoc.perl.org/functions/sprintf.html

Usage definition

```
#!/usr/bin/perl
use strict; use warnings; use Getopt::Long qw(GetOptions);

my $usage = <<"OPTIONS";
SYNOPSIS        Reformat numbers with Perl sprint
COMMAND         reformat.pl -n 12 -f 5 -w 10 -d 2
-n              Number to reformat
-d              Convert to decimals [Default: 0]
-w              Convert number to minimum width with padding zeroes [Default: 0]
-f              Convert scientific format to decimals
-e              Convert from decimal to scientific notation
OPTIONS
die "\n$usage\n\n" unless @ARGV;
```

Creating command lines switches with GetOptions();

```
my $num; my $dec = 0; my $wid = 0; my $flo; my $exp; my $new;
GetOptions('n=s' => \$num, 'd=i' => \$dec, 'w=i' => \$wid, 'f' => \$flo, 'e' => \$exp);
```

Printing reformatted numbers

```
print "\nOriginal number: $num\n";
if ($dec){$new = sprintf("%.${dec}f", $num); print "Decimal notation: $new\n";}
if ($wid){$new = sprintf("%0${wid}d", $num); print "Padding zeroes: $new\n";}
if (($wid) and ($dec)){$new = sprintf("%0${wid}.${dec}f", $num); print "Padding zeroes + decimals (min width of $wid): $new\n";}
if ($flo){$new = sprintf("%.${dec}f", $num); print "Scientific to decimal notation: $new\n";}
if ($exp){$new = sprintf("%.${dec}e", $num); print "Decimal to scientific notation: $new\n";}
print "\n";
```

Sprintf – Reformatting numbers

```
$number = sprintf("%.5f", $number);        ## Round up to 5 digits after decimal
$number = sprintf("%05d", $number);        ## Reformatting width to 5 char. (e.g. 00003)
$number = sprintf("%010.4f", $number);     ## Padding zeroes + floating (00003.1416 => %010.4f)
$number = sprintf("%e", $number);          ## Convert decimal notation to scientific number
$number = sprintf("%f", $number);          ## Convert scientific notation to decimal number
```

https://perldoc.perl.org/functions/split.html

Split and @arrays

http://perldoc.perl.org/functions/split.html

Splits by user-defined regular expressions

```
my @array = split("\t", $scalar);        ## Splits by tab (tsv)
my @array = split(",", $scalar);         ## Splits by comma (csv)
```

Usage definition

```
#!/usr/bin/perl
use strict; use warnings; use Getopt::Long qw(GetOptions);

my $usage =<<"OPTIONS"; ## Defining options
SYNOPSIS      print only desired columns from TSV file
COMMAND       split_columns.pl -c 1 5 12 -i file.tsv

-c (--columns)   ## Desired columns
-i  (--input)    ## Input file (in TSV format)
OPTIONS
die "\n$usage\n\n" unless @ARGV;
```

Creating command lines switches with GetOptions();

```
my @cols; my $input;
GetOptions(## Creating CMD line switches
    'c|columns=i@{1,}' => \@cols,
    'i|input=s' => \$input
);
```

Printing desired columns

```
open IN, "<", "$input" or die "Can't open file: $input\n";
while (my $line = <IN>){ ## Printing desired columns
    chomp $line;
    my @columns = split("\t", $line);
    for (0..$#cols - 1){
        my $num = $cols[$_] - 1; ## Arrays start at zero!
        print "$columns[$num]"."\t";
    }
    my $x = $cols[$#cols] - 1; ## Arrays start at zero!
    print "$columns[$x]\n";
}
```

Exercise 23 – Split and TSV/CSV tables

Create a script that:

1) Opens one or more TSV/CSV file (@files) from the CMD (-f *.tsv or -f *.csv)
2) Accepts the file type ($type) from the CMD (-type tsv)
3) Accepts the desired columns (@columns) from the CMD (e.g. -c 2 7 15)
4) Opens each file one by one (IN + OUT) and iterate through each line ## while loops
5) Chomps, then splits the line per tab (if tsv) or comma (if csv) in an array (@split)
6) Prints OUT the desired columns followed by a tab using foreach(@columns){…}
7) Prints OUT a new line character
8) Prints in the shell the number of columns found per file

```
## TIMTOWTDI with C++-style for loop
my @array;
for (my $x = 0; $x <= length$scalar; $x += 3){
	my $codon = substr($scalar, $x, 3);
	push (@array, $codon);
}
```

```
## TIMTOWTDI with unpack; much simpler
my @array = unpack ("(A3)*", $scalar);
```

Unpack and @arrays

https://perldoc.perl.org/perlpacktut.html

Unpacks by user-defined arguments

```
my @array = unpack ("(A3)*", $scalar);
```

ASCII characters | Width | Repeat till end of string

```
## Useful to deconstruct strings into smaller pieces

## Splits every 3 characters, great for translating mRNA
## to proteins with a codon usage table (%codons_to_aa).
```

Exercise 24 – Reordering/reformatting FASTAs

This is an odd FASTA file. Create a script that:

1) Reads one or more FASTA files from the CMD (-fa *.fasta) and a desired sequence length per line (-l 60) ## 60 nucleotides per line is standard
2) Puts the sequences in a hash (%sequences) with the sequence names as keys and the sequences themselves as values
3) Copies the names into an array as well (@names)
4) Reorder the sequence names alphabetically
5) Prints OUT the sequences alphabetically with the desired sequence length per line.

Exercise 25 – Translating DNA

Create a script that:

1) Reads one or more FASTA files from the CMD (-fa *.fasta)
2) Opens each file then iterate through each line ## while loops
3) Puts the sequences in a hash (%sequences) with the sequence names as keys and the sequences themselves as values
4) Unpacks each sequence by codons, *i.e.* triplets (A3;) and translate these codons into amino acids using a hash ## See slides #110 and 43-44
5) Writes OUT the translated sequences in amino acids.

Exercise 26 – Translating DNA again

Create a script that:

1) Reads one or more FASTA files from the CMD (-fa *.fasta)
2) Opens each file then iterate through each line ## while loops
3) Puts the sequences in a hash (%sequences) with the sequence names as keys and the sequences themselves as values
4) This time, let's use substrings and for loops instead of unpack.
5) Writes OUT the translated sequences in amino acids.

https://perldoc.perl.org/perlsub.html
https://perlmaven.com/subroutines-and-functions-in-perl

```
### Subroutines ###
sub samtools{
        print "Running samtools on $file.$fa.$mapper.sam...\n";
        system "$samtools"."samtools view -@ $threads -bS $file.$fa.$mapper.sam -o $file.$fa.bam";
        system "$samtools"."samtools sort -@ $threads -o $file.$fa.$mapper.bam $file.$fa.bam";
        system "$samtools"."samtools depth -aa $file.$fa.$mapper.bam > $file.$fa.$mapper.coverage"; ## Printing per base coverage i
        system "rm $file.$fa.bam"; ## Discarding unsorted BAM file
        $flagstat = `${samtools}samtools flagstat $file.$fa.$mapper.bam`;
}

sub variant{
        (my $passQC) = ($flagstat =~ /(\d+)\s+\+\s+\d+ in total/s); print "\nQC-passed reads mapping to the genome = $passQC\n\n";
        if ($passQC == 0){ ## Checking if BAM file is empty
                print "No QC-passed reads mapped to the genome; skipping variants calling\n\n";
                open VCF, ">$file.$fa.$mapper.$type.vcf"; print VCF '## No QC-passed reads mapped to the genome'."\n";
                close VCF;
        }
        else{ ## If not empty, proceed; empty BAM files create isssues when piping mpileup to some variant callers (e.g. VarScan2).
                print "Calling variants with $caller on $fasta...\n\n";
                if ($caller eq 'varscan2'){
                        if (($type eq 'snp')||($type eq 'indel')){system "$samtools"."samtools mpileup -f $fasta $file.$fa.$mapper.
                        elsif ($type eq 'both'){system "$samtools"."samtools mpileup -f $fasta $file.$fa.$mapper.bam | java -jar $v
                        else {print "\nERROR: Unrecognized variant type. Please use: snp, indel, or both\n\n"; exit;}
                }
                elsif ($caller eq 'bcftools'){
                        if ($type eq 'snp'){system "$samtools"."samtools mpileup -ugf $fasta $file.$fa.$mapper.bam | $bcftools"."bc
                        elsif ($type eq 'indel'){system "$samtools"."samtools mpileup -ugf $fasta $file.$fa.$mapper.bam | $bcftools
                        elsif ($type eq 'both'){system "$samtools"."samtools mpileup -ugf $fasta $file.$fa.$mapper.bam | $bcftools"
                        else {print "\nERROR: Unrecognized variant type. Please use: snp, indel, or both\n\n"; exit;}
                }
                elsif ($caller eq 'freebayes'){ ## single thread only, parallel version behaving wonky
                        system "$samtools"."samtools index $file.$fa.$mapper.bam";
                        system "$freebayes"."freebayes -f $fasta -p $ploidy $file.$fa.$mapper.bam > $file.$fa.$mapper.$type.vcf";
                        system "rm $file.$fa.$mapper.bam.bai";
```

get_SNPs.pl

Subroutines

```
sub Jeff {insert subroutine code here}   ## Defining a subroutine is easy
sub xyz {return $value}                  ## Invoke return to ensure that the subroutine returns the desired $value
Jeff(list, of, arguments);               ## Using a subroutine is easy too; arguments can be left blank if not needed
                                         ## $_[0] = list, $_[1] = of, $_[2] = arguments
&Jeff();                                 ## Older way of invoking subs (Perl < 5.0), not recommended
```

Subroutines do not have to be defined before they are invoked;
i.e. you can put them at the end of the script. It helps with code maintenance.

Exercise 27 – Of backticks and subs

Create a script that:

1) Contains a subroutine called pwd {} that:
   1) Creates a password with makepasswd of variable length $length
      ## Hint – Use backticks
   2) Chomps the password
   3) Replaces any single quotes (') characters by underscores (_) in the password
   4) Returns the desired password
2) Uses the subroutine to return a password of length $length in the shell; *e.g.* ./Ex_27.pl 36

Exercise 28 – Playing with sprintf

Create a script that:

1) Takes one of more number for the command line
   (*e.g.* -num 12 337.2 33 45)

2) Reformats them with the desired style (leading zeros, floating points) from the command line (*e.g.* -d 12 -f 3)

3) Can convert them from decimal to scientific and vice-versa

4) Prints the modified numbers in the shell

https://perldoc.perl.org/perldsc.html#HASHES-OF-ARRAYS
https://perlmaven.com/hash-of-arrays

%hash of @arrays

```
my %db                                  ## Powerful data structure; also very useful to parse tsv/csv files with split…
$db{yop}[0] = 'first';                  1st element (i.e. $yop[0]) of the array named @yop      ## Because these act as keys, each
$db{yop}[1] = 'second';                 2nd element (i.e. $yop[1]) of the array named @yop      ## array name must be unique
$db{yop} = ['first','second'];                                                                  ## TIMTOWTDI
print @{ $db{yop} }                                                                             ## Print the array @key from the hash
for (0 .. $#array) { $hash{$key}[$_] = $array[$_]; }                                            ## Populating a hash of array with a for loop
```

%hash of @arrays

An example

```
#!/usr/bin/perl
use strict; use warnings; use Getopt::Long qw(GetOptions);
```

Usage definition

```
my $usage =<<"OPTIONS"; ## Defining options
SYNOPSIS        Creates a hash from BLAST results;
                returns desired columns for each query
COMMAND         hash_of_array.pl -in blast.results -c 2 8 10 -q tig00000001

-in             BLAST results in TSV format (outfmt 6)
-c              columns desired
-q              queries
OPTIONS
die "\n$usage\n\n" unless @ARGV;
```

Creating command lines switches with GetOptions();

```
my $blast; my @columns; my @queries;
GetOptions( ## Creating CMD line switches from @ARGV
    'in=s' => \$blast,
    'c=i@{1,}' => \@columns,
    'q=s@{1,}' => \@queries
);
```

Creating the hash of arrays

```
open IN, "<", "$blast" or die "Can't open file: $blast\n";
my %blast_results; ## Creating database of results
while (my $line = <IN>){
    chomp $line;
    my @cols = split("\t", $line);
    for (0 .. $#cols) { $blast_results{$cols[0]}[$_] = $cols[$_]; }
    ## The line above will overwrite the content of existing keys! We could do a
    ## check with 'unless (exists $blast_results{$cols[0]})' to prevent problems
}
```

Doing something with the hash of arrays

```
for my $query (@queries) { ## Working through queries
    if (exists $blast_results{$query}){
        print "Query: $query \t";
        for (0..$#columns - 1){print "$blast_results{$query}[$columns[$_]-1]"."\t";}
        print "$blast_results{$query}[$columns[$#columns]-1]\n";
    }
}
```