Module 7

Introduction to Perl

Progamming for biology: an introduction to the Perl programming language

Introduction

When dealing with large datasets, we sometimes push the limits of desktop applications (e.g. spreadsheets), and we often find ourselves in situations where we have to repeat the same task or procedure for every item in our dataset. This repetitive process can often be tedious and time consuming. By learning to program in Perl we can write small programs, called scripts that allow us to easily automate these repetitive tasks.

Perl is a programming language that is widely used with nucleotide and protein sequences and for processing and managing large volumes of -– mostly text –- data.

Aims

The aim of this module is to present you with an introduction to the Perl programming language, guiding you through essential programming concepts. The first part of this module introduces the basic concepts of perl programming, illustrating these concepts using meaningful examples and exercises. The second part of the module introduces more advanced programming concepts, again illustrating them with relevant examples and exercises.

Many of the examples and exercises throughout this module are designed with sequence manipulation tasks in mind, and at the end of this module we hope you will be able to write a small program to perform a simple task. Like all modules, ‘if you don’t understand, please ask’. No-one is going to become a perl expert in a few hours; the overall purpose of the module is to give you a taste of what writing small programs in perl can do to automate and accelerate your analysis.

If at the end of this module you would like to learn more perl, please see the final part of this module in the Appendix which introduces more advanced perl concepts.

Perl Basics

What is a program? A program is a sequence of written statements or instructions that can be understood by a computer in order to perform a specified task.

What is a Perl program? A Perl program is a plain text file, containing statements or instructions written in the Perl programming language.

Perl stands for "Practical Extraction and Report Language"[1] which gives you an idea of what the aims were of its creator, Larry Wall. It has been widely used in the UNIX world for years for the automation of routine tasks, and for summarising large volumes of textual data.

[1] Less charitable people have said it stands for "Pathologically Eclectic Rubbish Lister"

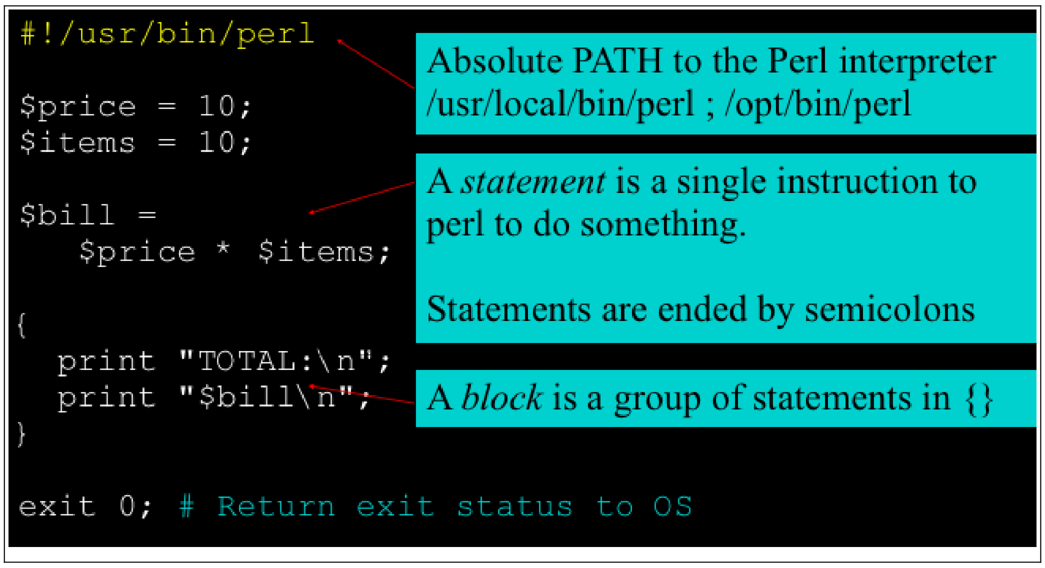
Writing perl programs

Any text editor can be used to write your perl program, but just remember that a word processor application (like Microsoft Word or OpenOffice) is **not** a text editor. A text editor doesn't allow you to format text (e.g. bold, italics, font sizes) and produces files in a plain (non-proprietary) format (.txt) that is readable in any computer.

Some freely available text editors

- Unix: vi, vim, nano, pico, gedit, mousepad, cream, emacs
- Windows: Textpad, PSPad, Notepad++
- Mac OS: Unix editors, TextWrangler

The basic structure of a perl program is shown in **Figure 1**. It is essentially just a list of *statements* which are the individual instructions for the computer to follow. Statements can be grouped into *blocks*. Blocks can contain other blocks.



```
#!/usr/bin/perl

$price = 10;
$items = 10;

$bill =
   $price * $items;

{
  print "TOTAL:\n";
  print "$bill\n";
}

exit 0; # Return exit status to OS
```

Absolute PATH to the Perl interpreter
/usr/local/bin/perl ; /opt/bin/perl
A *statement* is a single instruction to perl to do something.
Statements are ended by semicolons
A *block* is a group of statements in {}

Figure 1 A basic Perl script

Running perl programs

In order to get the computer to follow the instructions in a perl program you must execute (i.e. run) the perl program. In Unix, text files can be executed (i.e. run) as programs, provided they i) contain instructions in some language **and** the very first line of the text file starts with **#!** followed by the path to a program that can understand (interpret) the instructions. In our case, this will be the **perl** executable, that in most Unix systems is located in `/usr/bin/perl`.

To run (execute) a text file, either call it with perl:

```
perl program.pl
```

or give it execution privileges using `chmod`

```
chmod +x program.pl
```

and then execute it from the command line:

```
./program.pl
```

When saving a perl program it is standard practice to give it a .pl extension rather than a .txt extension.

Exercise 1

```
#!/usr/bin/perl

use strict;

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

Figure 2 Hello world

Type up the program above, save it as hello.pl and give it execution privileges. Then run it. Then modify the program so that it prints your name to the screen.

```
[ubuntu@ubuntu:$] ./hello.pl

Hello world!
```

```
[ubuntu@ubuntu:$] ./hello.pl

Hello Jacqui!
```

Commenting your program

Comments are used to explain what a particular piece of a program does and are important if you intend to modify, fix or look at the program again in the future. In a Perl program anything that follows a # sign is a comment and is ignored when the program is run.

```
print "Hello world!\n";  # This prints Hello world to the screen.
# This entire line is a comment.
```

Figure 3 Comments

Storing data and doing something with it

In Perl, as in any programming language, you use containers called *variables* to store data, change it, and access it later. Variables have names which you define:

```
$firstname, $lastname, @list_of_things_to_do, %people
```

Perl variables come in different forms, depending on the type of data they contain. The variables can be *scalars*, *arrays* or *hashes*. We will begin by looking at scalars. We will cover arrays and hashes later in this module.

Scalars

Scalars store data of any type (text, integers, floating point numbers). They are prefixed with **$**.

```perl
my $var = "GGATCCGGGACCAAAA"; # assign a "string"

my $val = 42; # assign a number

my ($a, $b, $c) = ("me", "my", "mine"); # assign all at once
```

Figure 4 Scalars

Quoting

Textual data in scalar variables needs to be placed in quotes, otherwise perl will think you are referring to perl commands. Different types of quotation marks have subtly different effects. The rules are the same as for the UNIX shell, and determine what happens to variable names within the quotes. In double quotes, the variable name is replaced by its value. In single quotes, it is not, and you see exactly what you put in the quotes.

The common term for a single textual piece of data is a *string*.

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

use strict;

my $name = 'Jacqui McQuillan';

print 'Hello $name\n'; # Probably not what you want
print "Hello $name\n"; # Better

print "\$name contains \"$name\"\n"; # Useful for debugging
```

Figure 5 Quoting

Exercise 2

Type up the program above, save it as quoting.pl and give it execution privileges. Then run it and inspect the output.

```
[ubuntu@ubuntu:$] ./quoting.pl
```

In the last example, you can see that characters like $ and ", which mean something special in this context, can be *escaped*, by using a backslash. In double quotes an escaped n character means 'newline'. As you can see this can get a bit unwieldy with all of the backslashes. Perl has an alternative syntax to double quotes to make this a bit clearer:

```
print qq{
\$name contains "$name"
}
```

Figure 6 qq quoting

Operators

Operators are small keywords which operate on variables; assign values, perform mathematical operations (addition, multiplication, division and subtraction), tests, and so on. Special operators like ++, –, +=, -=, /= and *= can also be used to manipulate a variables value without needing to repeat the variable in an equation.

```
# assignment operators, with numbers
my $a = 1;
$a += 2; # $a is now 3.  Shorthand for $a = $a + 2
$a *= 2; # $a is now 6

# now with strings
$a = "Me";
$a .= "Myself"; # $a is now "MeMyself"
$a .= "AndIrene"; # $a is now "MeMyselfAndIrene"
```

Figure 7 Perl Operators: assignment

For full details of what all the various perl operators are, see `man perlop`.

Exercise 3

Write a program to calculate depth of coverage for a lane of illumina data based on knowing the genome size, number of reads in the lane and the read length so that we know if we have enough coverage to call SNPs.

```
[ubuntu@ubuntu:$] ./calculate_coverage.pl
```

Arrays

Arrays are another type of variable and are used to store lists of items. They are prefixed with @ and can contain any number of scalar values.

The first element of the list is at position (index) zero of the array. You can count elements from the other end of the array by using negative numbers, starting from -1.

Note that you only use the @ prefix when referring to more than one element of the array. You use the $ prefix when referring to a single element of the array.[2]

You can also use the @ prefix when you wish to obtain a *slice* of the array; such as the elements at positions 3-5.

```perl
my @list = ("juan", "jose", "fred"); # assign 3 elements

print $list[0]; # print first element of @list
print $list[-1]; # print last element of @list

# Getting a slice:
my @some = @list[1..2]; # @some contains ("juan", "jose")

push @list, "roberto"; # adds string at the end
# now @list is ("juan", "jose", "fred", "roberto")

print $list[3]; # prints "roberto"

my $first = shift @list; # remove leftmost value and put in $first

my $last = pop @list; # remove rightmost value and put in $last
```

Figure 8 Arrays

Manipulating text

Perl provides a number of useful functions for manipulation of text. Using these functions called **split** and **join** it's easy to convert scalars into arrays and viceversa.

```perl
@values = split(/pattern/, $string);
```

```perl
$string = join("character(s)", @list);
```

The next example in **Figure 9** shows how to split a string into pieces (for example a sentence into words) using the perl function **split**. This is useful when reading data from a tab-delimited file (e.g. exported from Excel).

[2] This is not as confusing as it sounds; a single element of the array is a scalar quantity, so you refer to it using $ just like any scalar variable.

```perl
my $string = "Cel1980.1 ATG Hypothetical 2.54 High";

my @values = split(/ /, $string);
# @values are now ("Cell1980.1", "ATG", "Hypothetical", "2.54", "High")

print $values[3]; # would print 2.54

my ($id, @values) = split(/ /, $string);
# $id is "Cel1980.1"
# @values is now ("ATG", "Hypothetical", "2.54", "High")
```

Figure 9 From strings to lists and back again

Figure 10 gives examples of joining things back together again. Line 11 needs extra explanation: we're changing the value of one of perl's special built-in variables, `$,`, which is the *output record separator*. Its default value is " ", a single space. When you print an array, this variable is printed between each element of an array. So, in the above example, we're printing a tab character between each element of `@list`.

```perl
1  my @list = ("Cel1980.1", "ATG", "Hypothetical", "2.54", "High");
2
3  my $string = join("||", @list);
4  # $string is now "Cel1980.1||ATG||Hypothetical||2.54||High"
5
6  # in Perl if you want a TAB character you write \t in a double quoted string
7  my $another_string = join("\t", @list);
8  # now $another_string is TAB delimited
9
10 # Changing the output separator
11 $, = "\t"; print @list, "\n";
```

Figure 10 Convert a list of values into a string

Passing values to your program using the command line

Perl command line values or arguments are stored in a special array called **@ARGV**. So to access your program's command-line arguments you just need to read from that array. In the ARGV array, **$ARGV[0]** contains the first argument, **$ARGV[1]** contains the second argument and so on.

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

use strict;

my $first_name=$ARGV[0];
my $last_name=$ARGV[1];

print "Hello, $first_name $last_name\n";
```

Figure 11 Reading arguments from the command-line

Running the above program with the two arguments **Jacqui McQuillan** produces the following output.

```
[ubuntu@ubuntu:$] ./program.pl Jacqui McQuillam

Hello, Jacqui McQuillan
```

Exercise 4

Turn your program from Exercise 3 into a more general program that can work for any combination of genome size and sequencing data. Instead of hard coding the values for genome size, number of reads, and read length - we now want to make the program accept the values from the command line and print the depth of coverage to the screen.

```
[ubuntu@ubuntu:$] ./calculate_coverage.pl lane1 500000 75 4000000

lane1 9.375X
```

Loops and iteration

As in other programming languages, Perl provides functions to repeat tasks, iterating instructions within loops, performing the same analysis on a list of items. In **Figure 12**, **expression** refers to some code that evaluates to *true* or *false*, while **execute block** refers to a number of statements that are executed while/until the condition specified in the expression is not met.

```perl
while (expression) { execute code within this block }
do { execute this code of block } until ( expression )

# for (initial, expression, increment) { execute block }
for ( $i = 0; $i < 100; $i = $i + 1) { print "$i\n" }
# start at 0, continue while $i < 100,
# increment $i by 1 in each iteration

# a special iteration for arrays
foreach $item (@list) { execute block }
# in each iteration $item will contain the next element of @list
```

Figure 12 Iterations, loops

Reading and writing files

Perl variables only last as long as the program is running. You will usually want to read input from disk files into your variables, and you will need to write output back to disk files.

To read from files and write to files, you declare a *file handle* (a name) and associate it with your file. Then you refer to this handle every time you want to read from or write to your file. The special operator **<>** is used to read from a handle (i.e. <HANDLE>), while **print** is used to write to a handle (i.e. print HANDLE).

```
# reading from files
open(MYHANDLE, "/home/fernan/somefile.txt") or
  die "Could not open file: $!\n";
while ( $line = <MYHANDLE> ) {
 # read the file one line at a time,
 # storing the contents of each line into $line
 # use the data in $line to do something
close MYHANDLE;
}

# write to files
open(MYOUTPUTHANDLE, ">/home/fernan/somefile.txt") or
  die "Could not open file: $!\n";
print MYOUTPUTHANDLE "Hello there!";
close MYOUTPUTHANDLE;

# append to files (don't overwrite the contents of somefile.txt)
open(MYOUTPUTHANDLE, ">>/home/fernan/somefile.txt") or
  die "Could not open file: $!\n";
print MYOUTPUTHANDLE "Hello there!";
close MYOUTPUTHANDLE;
```

Figure 13 Reading from files, writing to files

At this point we need a quick aside to deal with the matter of error handling. When you use perl functions which ask the operating system to do something for you (such as opening a file), it is possible that the operation will fail. The file might not be there, or you might not have permission to read or write the file. It's important to detect when these errors occur, and to inform the user. Otherwise, your program will not produce the right results, and you won't know why. All functions like `open()` actually return a true or false value, depending on whether they work or not. In **Figure 13**, the second part of each `open()` statement, beginning with the keyword `or`, detects if the `open()` fails, and if it does it calls the `die` function. `die()` causes the program to stop running, and print an error message. In the error message we have included another of perl's special built-in variables, `$!`. Whenever a function like `open()` fails, perl puts the reason for the failure in `$!`. So, if you try the first example above, you will receive the error message:

```
Could not open file:  No such file or directory
```

It is good programming practice to always check for errors in this way. It may feel like more work, but if you get into the habit, it will save you many hours of hunting for errors in your programs in the future.

Perl provides a number of special handles that are always open and available (you don't need to *open* them). These are:

- **STDIN** Standard input, usually the keyboard, but in Unix you can redirect it. to read from a file or from the left hand side of a pipe)
- **STDOUT** Standard output, usually the screen, but you can redirect this stream to a file
- **STDERR** Standard error, similar to STDOUT, but where functions like `die()` and `warn()` send their error messages, to keep them separate from real output.

```
# e.g. in a program called filter.pl
while (<STDIN>) { # read from the keyboard or from a pipe }

# then we use filter.pl like this
# 'cat sequences.fasta | filter.pl' or 'filter.pl < sequences.fasta'

# we print data to STDOUT and warnings/errors to STDERR
print STDOUT "MW: 325.08 kDa", "\n", "pI: 9.54", "\n", "Length: 2954 aa";
print STDERR "Warning: sequence length is zero!";
warn "Warning: sequence length is zero!"; # Better equivalent

# then we can send each to a separate file
# filter.pl > data.txt 2> errors.txt
```

Figure 14 Special handles

Exercise 5

Now we are getting several lanes of sequencing data per week and would like to be able give the program a spreadsheet of values for various different illumina lanes and genome size. Modify your program from Exercise 4 so that it reads in the name of a spreadsheet (tab delimted file) that contains a list of lanes, number of reads, read length and genome size. Your program should print the results to the screen for each lane.

```
[ubuntu@ubuntu:$] ./genome_coverage.pl mylanes.tsv
```

Exercise 6

Now we want our program to output another spreadsheet (tab delimited file) so that we can email the results of the program to our boss as a summary of how much sequencing data has been generated each week. Modify your program from Exercise 5 so that it prints the results to a file instead of to the screen. The name of the output file should be specified on the command line.

```
[ubuntu@ubuntu:$] ./genome_coverage.pl mylanes.tsv myresults.tsv
```

Decision Statements

Sometimes you will want to perform different tasks depending on whether a condition is true or false. In Perl this can be achieved with the keyword, `if`. The `if` statement consists of a condition that Perl evaluates, and a block of code in curly brackets that Perl runs if the condition evaluates to true (`if-block`) and an optional block of code that Perl runs if the condition evaluates to false (`else-block`).

```
if ( CONDITION ){
  #if condition is true
}
else{
  #if condition is false
}
```

Figure 15 Perl if statement

Both logical and comaprison operators can be used in combination with with perl `if` statements as shown in **Figure 16**.

```
# control operators
# logical AND (&&), OR (||), NOT (!)
if ( $a && $b ) { # do something }
if ( $mw > 100 || $pi < 9 ) { ... }
if ( ! defined $c ) { ... }

# comparison operators
# numerical <  >  <= >= != ==
# strings   lt gt le ge ne eq
if ( $a == 4 ) { # do something }
if ( $b eq "ATG" ) { ... }
```

Figure 16 Perl Operators: logic and control

The logical operators come in two flavours, one that looks mathematical, and one that looks English. You can use either; the only difference is their *precedence*. For example, you use either `&&` or the English word `and`.

The comparison operators also exist in mathematical and English versions, but this time they **are** different. The mathematical version compares variables based on their numerical value, whereas the English version compare based on alphabetical order:

```
if ('Alice' lt 'Bob') {
  print "Alice comes before Bob, alphabetically\n";
}
```

Figure 17 Text comparison

A trap for the unwary:

It's a very common error to accidentally use = when you meant ==, because both are valid in an `if()` statement, but strange things can happen if you use the wrong one:

```
my $a = 1;
if ($a = 2) { # We meant $a == 2 here, of course
  print "1 equals 2!\n"
}
```

Figure 18 = is not the same as ==

Why does this code do what it does? Because `$a = 2` does two things. First, it assigns the value of 2 to $a, which we didn't want it to do. But that expression also *returns* a value of 2 to the `if()`. Remember that non-zero values are always logically true in perl, so it then executes the print statement.

Exercise 7

Write a program that counts the number of genes in a GFF file. The name of the GFF file should be specified on the command line and the result should be printed to the screen.

```
[ubuntu@ubuntu:$] ./count_genes.pl augustus.gff
```

Hashes

Hashes or associative arrays in Perl store values in pairs (key <=> value), and are prefixed with %

```perl
my %me = (
  name => "Jacqui",
  age => 31,
  loves => "Perl",
); # create a hash with 3 key/value pairs

print $me{name}; # print value associated with 'name'

$me{born} = "Ireland"; # add a new key-value pair
$me{eyes} = "Green";
```

Figure 19 Hashes

Beware: Referring to a key which does not already exist in the hash will make it magically spring into existence. This can be a cause of subtle bugs, which can be hard to find. If you need to test whether a key already exists in a hash, you use the `exists()` function, as shown in **Figure 20**

```perl
if ($me{age}) { print "This could cause bugs\n" }

# The right way:
if (exists($me{age})) { print "We know your age\n" }
```

Figure 20 Using exists()

Exercise 8

Write a program that counts the number of each feature type in a GFF file. The name of the GFF file should be specified on the command line and the results printed to the screen. The results should consist of a list of all feature types (gene, transcript, CDS, etc.) present in the GFF and the number of each feature type present in the GFF file.

```
[ubuntu@ubuntu:$] ./count_features.pl augustus.gff

gene : 378

transcript : ...
```

Text processing operators

Perl also offers very flexible operators, **=~** and **!~**, that don't use literal strings, but use patterns. These patterns known as *regular expressions*, or *regexes* for short.[3] Regular expressions may match more than one possible string (think about allowing mismatches), and provide a very powerful tool for matching text. Indeed, they really are the heart of perl.

There are three operators that go with the `=~` operator:

- `m/regex/` or `/regex/` tests whether the regex can be matched in the string
- `s/regex/replacement/` finds a match of the regex
- `tr/characters/replacements/` finds individual characters and substitutes them for others.[4]

```perl
my $seq = "ATGGCGTAACGAATTCAATGAACGTTTGG";
# match against an EcoRI site
if ( $seq =~ /GAATTC/ ) { # do something }
# match against EcoRI or BamHI sites
if ( $seq =~ /G[AG]AT[TC]C/ ) { # do something }
# don't match runs of polypyrimidines (T or C)
if ( $seq !~ /[TC]+/ ) { # do something }

# substitute (find/replace)
$seq =~ s/GAATTC/GGATCC/; # s/EcoRI/BamHI/

# Reverse complement a sequence
$seq =~ tr/[ACGTacgt]/[TGCAtgca]/;
$seq = reverse($seq);
```

Figure 21 Match and substitute operators

The above are very simple examples. For full details on these operators read the `perlop` manpage, and a description of regular expressions can be found in the `perlre` manpage.

Regular expressions can be very complex and used for really very complex parsing of data. However, if you're trying to parse a file format where records are spread over multiple lines, such as XML or HTML, regular expressions can become difficult to get right. Fortunately, perl modules to parse most common formats can be found on CPAN. CPAN, stands for the Comprehensive Perl Archive Network and is an archive of freely available software modules written in Perl.

Exercise 9

Write a program that calculates the GC content of a DNA sequence in a FASTA file and prints the result to the screen.

```
[ubuntu@ubuntu:$] ./gc_content.pl Transfer.ordered_Pf3D7_05.final.fa
```

[3] You met this term in Module 1: Basic UNIX. Remember that **grep** stands for *global regular expression print*?

[4] Just like the UNIX `tr` command, hence the name of the operator

Exercise 10 - Transform the output of Glimmer and visualize it in Artemis.

Glimmer is a gene prediction program for prokaryotic genomes (a *gene finder*). It can be used to identify genes in a newly sequenced genome. Before attempting to find genes, Glimmer needs to be trained with a set of known (validated) protein coding genes from the target organism/genome.

Predictions made by Glimmer can't be visualized in Artemis without reformatting the output. In this exercise, you will analyze the output of Glimmer, and write a program to convert the output (producing a new file) so that it can be read by Artemis.

Features can be read by Artemis in the following format: the Generic Feature Format (GFF), which is a plain text (tab-delimited) file (**Figure 22**).

```
clari    source    gene    1     1329     .     +     .     ID=dnaA
clari    source    gene    1462     2529     .     +     . ID=dnaN
clari    source    gene    2539     4857     .     +     . ID=gyrB
```

Figure 22 Features in GFF format

Write a program to convert the output of Glimmer into GFF so that it can be visualized in Artemis. Check that all predictions are displayed correctly (i.e. in their corresponding strand, frame, location).

Input file: mtu.glimmer

Discuss the solution (your program) and compare it with the programs written by others.